**MASTER IN COMPUTER SCIENCE PROFESSIONAL FOCUS IN DATA SCIENCE**

# Intelligent Code Completion Using Distributed Representation of Code

WEYSSOW, Martin

*Award date:*
2020

*Awarding institution:*
University of Namur

UNIVERSITÉ DE NAMUR

FACULTÉ D'INFORMATIQUE

**Intelligent Code Completion Using Distributed Representation of Code**

Martin WEYSSOW

RUE GRANDGAGNAGE, 21 ● B-5000 NAMUR(BELGIUM)

UNIVERSITÉ DE NAMUR
Faculté d'informatique
Année académique 2019–2020

# Intelligent Code Completion Using Distributed Representation of Code

Martin WEYSSOW


Maître de stage : Houari Sahraoui, Université de Montréal, CA

Promoteur : Benoît Vanderose
(Signature pour approbation du dépôt - REE art. 40)

Co-promoteur : Benoît Frénay

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

# DEDICATION AND ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my internship supervisor, Prof. Houari Sahraoui who received me at the University of Montreal in Canada. He has been an incredible supervisor and helped me to complete this thesis. On top of that, Prof. Sahraoui has taught me a lot about conducting research in software engineering and I cannot be grateful enough to him for offering me the opportunity to experience a year of research in the beautiful city of Montreal.

I would also like to thank my two supervisors, Prof. Benoît Vanderose and Prof. Benoît Frénay for making my internship at the University of Montreal possible. I also thank them for their support during my internship as well as their contribution to research works in collaboration with Prof. Houari Sahraoui.

My gratitude also goes to all the people from the GEODES lab of the University of Montreal for their warm welcome.

Finally, I would like to express my deepest gratitude to my parents, family and friends for their support during my studies and especially during my year spent in Montreal. A great part of the success I have had during my studies are due to their support throughout these last years and I will be forever grateful to them.

## ABSTRACT

Code completion is an important feature of integrated development environments (IDEs). It allows developers to produce code faster, especially novice ones who are not fully familiar with APIs and others' code. Previous works on code completion have mainly exploited static type systems of programming languages or code history of the project under development or of other projects using common APIs. In this work, we present a novel approach for improving current function-calls completion tools by learning from independent code repositories, using well-known natural language processing models that can learn vector representation of source code (*code embeddings*). Our models are not trained on historical data of specific projects. Instead, our approach allows to learn high-level concepts and their relationships present among thousands of projects. As a consequence, the resulting system is able to provide general suggestions that are not specific to particular projects or APIs. Additionally, by taking into account the context of the call to complete, our approach suggests function calls relevant to that context. We evaluated our approach on a set of open-source projects unseen during the training. The results show that the use of the trained model along with a code suggestion plug-in based on static type analysis improves significantly the correctness of the completion suggestions.

# RÉSUMÉ

La complétion automatique est une fonctionnalité importante des environnements de développement intégrés (IDEs). Celle-ci permet aux développeurs de produire du code plus rapidement, et est particulièrement utile pour les novices qui ne sont pas familiarisés avec certaines APIs et les codes d'autres développeurs. Les travaux précédents sur la complétion automatique ont principalement exploité des techniques d'analyse statique des langages de programmation ou des historiques de code du projet en développement ou d'autres projets utilisant des APIs standards. Dans ce travail, nous présentons une approche originale pour améliorer les moteurs de complétion automatique d'appels de fonctions en apprenant depuis des répertoires de codes indépendants, en utilisant des modèles bien connus de traitement automatique des langues naturelles qui permettent d'apprendre des représentations vectorielles du code source (*code embeddings*). Nos modèles ne sont pas entraînés sur des données historiques de projets spécifiques. Au lieu de cela, notre approche permet d'apprendre des concepts de haut niveaux et leurs relations parmi des milliers de projets. Par conséquent, le système résultant est capable de fournir des suggestions générales qui ne sont pas spécifiques à des projets ou des APIs. De plus, en prenant compte le contexte de l'appel à prédire, notre approche suggère des appels pertinents pour ce contexte. Nous évaluons notre approche sur un ensemble de projets open-source non-vus durant l'entraînement. Les résultats montrent que l'utilisation d'un modèle entraîné en conjonction avec un plug-in de suggestion de code basé sur l'analyse statique du code améliore significativement l'exactitude des suggestions de ce dernier.

# TABLE OF CONTENTS

# List of Tables

# List of Figures

CHAPTER 1

# INTRODUCTION

## Context and Problematic

Nowadays, developers rely on features provided by modern Integrated Development Environments (IDEs) to ease their cognitive load and increase their productivity. One purpose of these features is to avoid asking developers to provide information that can be inferred from the available data sources and the current development context [Murphy, 2019]. Among these features, code completion is one of the most widely used by, among others, Java developers in Eclipse [Murphy et al., 2006]. Code completion helps developers to write code faster by providing a list of suggestions of possible elements, such as function calls, pertinent to a given context.

There have been a lot of research contributions that attempt to improve code completion systems. Early learning-based approaches focused on completion, specifically for APIs by leverage historical or context data about the system under development [Bruch et al., 2009; Proksch et al., 2015]. From another perspective, work has been done to exploit natural language modeling for, among other tasks, code completion, based on the idea of code naturalness [Hellendoorn and Devanbu, 2017; Hindle et al., 2012; Tu et al., 2014]. More recently, other approaches have targeted AST representations of the code to perform the APIs calls completion [Bhoopchand et al., 2016; Li et al., 2017; Nguyen and Nguyen, 2015; Svyatkovskiy et al., 2020, 2019]. In general, the above-mentioned works exploit historical data from the projects used during the evaluation of the system and/or evaluate their systems on specific APIs completion. In the first case, the approaches are not applicable to new projects or projects with short histories, whereas, in the second case, the objective is to predict the calls to APIs' methods. Although the obtained results are convincing, these approaches have shown to be efficient only for popular libraries.

## Approach

In this work, we propose a novel approach for improving function calls completion by learning models from independent code repositories. Our goal is to allow call completion not only with API functions, but also those of the project under development. More specifically, we consider each method as a natural text paragraph containing a sequence of function calls. Then, using a well-known word embedding model, we learn vector representation of variable-length sequences of these paragraphs. Our approach is based on the assumption that there exist recurring patterns of function-call sequences among the code repositories and that these patterns capture semantics about higher-level concepts. Our approach is intended to abstract these high-level concepts and use them to improve function-call completion by comparing the call site context with the huge amount of contexts learned from the repositories. We explore two ways for using the learned models for function-call completion. The first way is to build a suggestion list from scratch without taking into account the knowledge about the project under development, *i.e.,* the functions that can be called in the project. The alternative way is to exploit the learned model to reorder the suggestion list proposed by a given typing-based code completion system.

To evaluate the proposed approach, we used a corpus of more than 14,000 Java projects from which we extracted more than 10 millions function sequences to train our models. To test our completion strategies, we selected 10 projects, not considered for the training, and having more than 160.000 call sites to complete. The results of our evaluation show, on the one hand, that the from-scratch strategy has completion results close to those of Eclipse's content assist, and that the precision increases with the size of the context, *i.e.,* the number of calls previously performed before the call site. However, these results are insufficient considering the effort made to train the models. On the other hand, the reordering strategy improved the completion precision of Eclipse, for 9 of the 10 projects, by up to 135% reaching 85% of Precision@10. Only the smallest project had lower scores because of the specific vocabulary not seen in the models. To cope with this situation, we explored, with a relative success, the use of subtokens rather that full-function names. Finally, we found that it takes between 700 ms and 800 ms, on average, to produce completion suggestions for a call site. This makes our approach usable in a real programming setting.

## Research Questions

Our contribution is articulated around several research questions. Each of them is formulated in more details in the Section **4.3** about the evaluation setup of Chapter **4**.
Our first two research questions intend to validate the rational behind our approach as

well as our hypothesis before the experiments on code completion:

- **RQ1 [Replication]:** ***How repetitive and predictable are function sequences in source code?***
- **RQ2:** ***Are paragraph vector embedding models capable of capturing concepts from the code?***

The idea underlying these two research questions is to perform qualitative evaluations of our models that are independent from the code completion task (*i.e., intrinsic evaluations*). These experiments allow us to determine whether there is a positive correlation between the effectiveness of our systems on code completion and some qualitative analysis of these. We answer to both RQs in Sections **4.4.1** and **4.4.2**.

Then, we address the RQs about the code completion experiments:

- **RQ3:** ***Using the paragraph vector model, can we accurately suggest function invocations given a context?***
- **RQ4:** ***Can we use the paragraph vector model in order to improve the suggestion ranking made by Eclipse's content assist plug-in?***

RQ3 and RQ4 enable us to evaluate our systems in two experimental setups that are meant to be realistic *w.r.t* the usage of an integrated development environment. We answer to these RQs in Sections **4.4.3** and **4.4.4**.

## Structure

The rest of the thesis is structured as follows. In Chapter **2**, we introduce an overview of natural language processing, basic language models and distributional approaches for learning vector representations of words. Chapter **3** introduces the state-of-the-art in code completion and its limitations. Chapter **4** presents the core of this thesis and covers our approach, the experimental setup and the results of our experiments. Finally, details on future work opportunities and a conclusion are presented in Chapters **5** and **6**.

# Part I

# Background and State-of-the-art

CHAPTER 2

# NATURAL LANGUAGE PROCESSING AND SOFTWARE ENGINEERING

In this thesis, we address the problematic of code completion using natural language processing (**NLP**) and word embedding models trained on a huge amount of Java projects. This chapter serves a twofold purpose. Firstly, it intends to give some insight about a set of techniques commonly used in NLP applied to source code. And secondly, it aims to describe the machine learning (**ML**) models that we use in our experiments in order to learn meaningful representations of source code artifacts. These models allow us to build a code completion engine and enhance the effectiveness of an existing one based on static type analysis of the code.

## 2.1 Introduction

Natural language processing aims at making computers understand natural language texts. It provides tools and techniques to process and analyze large amount of textual data as leverage to some downstream tasks. Among these tasks, we find machine translation that consists of translating texts from a source language to a target language. Another task is information retrieval that intends to find relevant information to an information need from a large document set (*i.e., search engines*). Both of these tasks can take advantage of NLP and machine learning techniques in order to improve their effectiveness. For instance, estimating a language model on a search engine corpus (*i.e., web pages*) can help to learn linguistic regularities within the web pages and better match the users' queries.

With a view to using NLP and ML, it is required to have a relatively large corpus of texts at one's disposal and suitable for the task. In a classical NLP pipeline, cleaning techniques are used to preprocess the data before performing a learning task on the corpus. A common learning task consists of learning useful word representations that

will support a downstream task. There exists two main kind of approaches to estimate or learn these representations : **(1)** statistical approaches (*i.e., bag-of-words, TF-IDF, n-grams language models*) where probabilities are assigned to words or sequences of words based on statistics of these in a given training corpus, **(2)** learning-based approaches where a machine learning model attempts to learn vector representations of words or sequences (*i.e., word embedding*) that capture syntactical and semantics information of words within a training corpus.

In Section **2.2**, we present NLP techniques commonly used to clean and normalize a corpus. In Section **2.3**, we show how we can use these techniques on source code. Then, we emphasis the utilization of a statistical approach to estimate the probability of a corpus with $n$-gram language models in Section **2.4**. We address *(deep)* neural approaches for word and sequence representations in Section **2.5** and we close this chapter with a conclusion in Section **2.6**.

## 2.2 Natural Language Processing Techniques

We present a set of techniques largely used in NLP pipelines and illustrate their usage on some examples. The goal of these techniques is to prepare and clean textual data in order to keep meaningful information from the text.

### Tokenization

Tokenization is usually the first task that is performed when working with textual data. It consists of cutting sentences into tokens. Usually, a tokenizer uses space and punctuations to cut the text. Each token is then a candidate for further processing. Here is an example of the tokenization of a sentence using space and punctuation:

- Input: “Friends, Romans and Countrymen”
- Output: "Friends", "Romans", "and", "Countryman"

In this basic example, the tokenization process is straightforward because there is no ambiguous tokens. But, in practice, it is not always the case. Here are some examples where we need word-sense disambiguation:

- Input: “United Kingdom” → one or two words ?
- Input: "aren't" → "arent" ? "aren" "'t" ? "aren" "t" ?
- Input: "192.168.1.1" → it depends on the task and the data

The simplest strategy consists of building a dictionary with words that need to be kept as is or that require a special treatment. Once the corpus tokenized, we can build a vocabulary that corresponds to the set of tokens in the corpus. The next cleaning techniques essentially help reducing the vocabulary space and making the text more uniform.

### Normalization

Depending on the corpus we are working with, we need to normalize some terms that have the same meaning but that appear in several forms. For example, we need to match "U.S.A" and "USA" as a wholesome token.
We may also deal with numerical tokens (*e.g., IP addresses, numbers, dates*) and special tokens (*e.g., ..., !!!, ?!*). These tokens can either be deleted because they usually carry non-essential information or they can be normalized as follows:

- Input: "192.168.1.1" → "<IP_ADDRESS>"
- Input: "Sept 28, 2020" → "<DATE>"
- Input: "!!!" or "!!" → "<STRONG_!>"

In that way, we keep some information about the text, reduce the list of tokens and summarize a lot of them into a unique one. There exist a lot of other cases where there is a need to normalize the text depending on the task to be performed. Normalization can be a non-ending process, therefore we usually choose to normalize phenomena that appear frequently in a corpus.

### Stopwords Removal

Removing words that do not bear useful information is also a process that is commonly performed in NLP. For instance, in english, these stopwords are "*of, in, the, with, I, she, ...*". These are usually function words that appear often in a corpus. There exist a few standard lists of stopwords commonly used that can be adapted manually for a specific corpus.

### Stemming and Lemmatization

The goal of stemming is to create a standard representation for terms that bear similar meaning but that have different forms. Stemming algorithms remove endings of words in order to obtain a single form. For example :

- Input: computer, compute, computes, computing, computed, ...
- Output: comput

There exist several stemming algorithms that also depend on the corpus' language. For instance, Porter algorithm stands among the most known stemming algorithm [Porter, 1980]. Porter stemmer makes less errors compared to other stemming algorithm and has been implemented for several languages [Ismailov et al., 2016].

Lemmatization is a process similar to stemming that generates roots of words. The difference is that a word that has been stemmed might not be an existing word because

stemming algorithms are based on heuristics whereas, lemmatization yields standard form of words that are existing words. Both techniques allow to reduce the space of the vocabulary and make the text more consistent.

## 2.3 Natural Language Processing and Source Code

A source code artifact is similar to a corpus of texts. In the same way as for natural language data, it consists of sequences of tokens on which we can perform NLP techniques discussed in the previous section. A source code corpus is usually a collection of projects cut into blocks of code. Each block can then be seen as a sequence of tokens or a sentence. To illustrate the usage of NLP techniques on source code, lets us consider a corpus of Java code cut into blocks of methods:

```
1 // ...
2 public long size() throws IOException {
3   if (!file.isFile()) {
4     throw new FileNotFoundException(file.toString());
5   }
6   return file.length()
7 }
```

**Listing 2.1: Motivating example**

In order to extract the tokens from this block of code, the most efficient approach is to use an abstract syntax tree (**AST**) parser. In fact, using a simple white space tokenizer would not separate "(!$file.isFile()$)" and defining special tokenization rules would be too costly and inaccurate. By using an AST parser, we obtain the following sequence:

$$"public", "long", "size", "(", ")", "throws", "IOException", "\{", "if", ...$$

The extracted sequences can then be used to estimate a language model or train a word embedding model. But, as for natural language texts, we may want to perform further preprocessing on the code to reduce the noise in the data and thus better support the learning task. It is important to mention that the preprocessing pipeline needs to be carefully designed depending on the end-task to be performed.

### Syntax Tokens Filtering

As discussed in above Section **2.2**, stopwords do not bear useful information in a text. This notion of stopwords in source code artifacts has been introduced in a recent work and refers to the syntactical tokens (*e.g.,{}, (), if, ...*) [Rahman et al., 2019]. *Rahman et al.* have shown that syntax tokens make the code artificially repetitive. In fact, these tokens are part of the language specification and appear very frequently in the code. Therefore, ignoring

these tokens can be useful for some tasks (*e.g., code summarization, code completion*). Similarly, tokens such as "*public*", "*throws*", "*return*" could be filtered to obtain only tokens referring to identifier names which carry a great part of the code's semantic.

### Tokenization of Identifiers

The internal structure of identifiers names (*e.g., function names and variable names*) is generally composed of concatenation of words. For instance, in Listing **2.1**, the function call "*isFile*" is made of the two subwords "*is*" and "*file*". The words contained within the identifier are called **subtokens**. Previous works have shown that the tokenization of identifiers is useful for a code recommender to predict **out-of-vocabulary** (**OOV**) tokens (*i.e., tokens that does not appear during the training phase of a model*) [Allamanis et al., 2015; Karampatsis et al., 2020; Svyatkovskiy et al., 2020].
In fact, it is more likely that an identifier such as "*zoomsOutALittleAfterNotification-SoAllSpecialControlsAreInitializedWhenItHappend*" will not appear in a training corpus. By considering subtokens of identifiers, a model could learn the internal structure of identifiers and recommend out-of-vocabulary identifiers by concatenating subtokens. Moreover, splitting identifiers into subtokens reduces drastically the size of the vocabulary thanks to their redundancy. This reduction can speed up the training process of a ML model and reduces its storage size on a physical device.

In practice, the tokenization of identifiers has to be done carefully, depending on the programming language we are working with. For instance, in Java, the convention is to use camel case to separate subtokens, while in Python, most programmers use the underscore. Therefore, the tokenizer must be adapted accordingly.

### Identifiers Roots

When considering the subtokens of identifiers, we may chose to preprocess the code one step further. The subtokens contained within an identifier are usually actual english words. For example, a function named "*getNames*" contains two english words that can be stemmed or lemmatized.

### Full Example

Lets consider the example of Listing **2.1**. After preprocessing the code with syntax filtering and camel case tokenization of identifiers, we get the following sequence:

$$
\begin{aligned}
&"size", "io", "exception", "!", "file", "is", "file", "file",\\
&"not", "found", "exception", "file", "to", "string"\\
&"file", "length"
\end{aligned}
$$

In that way, we obtain data containing meaningful tokens. A corpus composed of such sequences can then be fed into a statistical language model or a machine learning model to estimate or learn word/sequence representations.
In the next section, we discuss $n$-gram language models and illustrate their utilization for source code modeling.

## 2.4 $n$-gram Language Models

Language models (**LMs**) assign probabilities to sequences of words. A training phase is performed using a set of documents (*i.e., a training corpus*) generally written in the same language. Then, the regularities captured by a LM in the training corpus can be used as leverage for performing an extrinsic task (*e.g., speech recognition, spelling correction, text generation, ...*). In this section, we discuss the $n$-gram model which is the simplest form of language model.

### 2.4.1 Motivating Example

Lets consider a predictive text tool whose role is to predict the next word a user is going to type on his smartphone. The tool can take advantage of the previously typed words in order to make the prediction more accurate. The goal of the task is to find the word $w$ that maximizes the probability $P(w \mid h)$, *i.e.,* the probability of $w$ conditioned on an history $h$ made of the previous words. For instance, lets us consider that the user is writing the following sentence : "*Hello, I am on my way back*". The predictive text tool needs to compute:

$$\arg\max_{w} P(w \mid h = \textit{Hello, I am on my way back}). \tag{2.1}$$

A simple approach to determine this probability is to count the occurrence of the sequence "*Hello, I am on my way back*" in the given training corpus and find the word $w$ that most often follows this sequence (*i.e., maximum likelihood estimation*). While this approach may be doable for this simple example, it is not the case in practice. In fact, for a more complex history sequence, it will be unlikely that the sequence appears in the training corpus. Therefore, computing $P(w|h)$ would not be feasible. In the next section, we investigate how to compute these probabilities with more ease using $n$-gram language models.

### 2.4.2 Estimating the Probability of a Sequence

Considering a word sequence $w_1, w_2, ..., w_n$, a LM assigns a probability $P(w_1, w_2, ..., w_n)$ to the sequence. We can estimate the probability of the whole sequence using the chain

rule of probability:

$$P(w_1, w_2, ..., w_n) = P(w_1)P(w_2|w_1)...P(w_n|w_{n-1}, ..., w_2, w_1)$$

Such a probability is still hard to compute because as we have seen in the previous section, long sequences of words are usually not observed in a training corpus. Therefore, we use $n$-gram language models to approximate $P(w_1, w_2, ..., w_n)$.

$n$-gram language models assign a probability to a word $w$ given an history of size $n-1$. $n$-gram LMs assume that the occurrence of a word depends only on the previous words. In other words, a $n$-gram model is a **Markovian approximation** of order $n-1$:

$$P(\mathbf{w}_1^n = w_1, w_2, ..., w_n) \approx \prod_{k=1}^{n} P(w_k|w_{k-n+1}^{k-1})$$

For instance, given a bigram (*i.e., 2-gram*) model, the probability of a word $P(w_k \mid w_{k-1})$ depends only on the previous word $w_{k-1}$. Given the example of equation **2.1**, when using a bigram model, we obtain:

$$P(\mathbf{w} \mid h = \textit{Hello, I am on my way back}) \approx P(\mathbf{w} \mid h = \textit{back})$$

As we increase the order of the $n$-gram model, the probability of the sequence gains in acuteness. Nevertheless, usually we choose a small value for $n$ (*i.e.,* $n \in [2,5]$) to simplify the counts and avoid zero probabilities.

As we have seen before, the simplest approach to estimate these word probabilities is by computing a maximum likelihood estimation (**MLE**) over the raw counts of $n$-grams in the corpus. For a bigram model, the probability of a word $w_n$ given the history $w_{n-1}$ is:

$$P(\mathbf{w}_n|w_{n-1}) = \frac{|w_{n-1}w_n|}{\sum_w |w_{n-1}w|}$$

In practice, we do not directly use MLE to avoid the model to assign zero probabilities to unseen sequences of words. Instead, the usage of smoothing techniques allows to assign a part of the total probability mass to unseen $n$-grams as depicted in Figure **2.1**.

### 2.4.3 Smoothing Techniques

#### Laplace – additive smoothing

**Laplace smoothing** is a very simple technique that consists of adding counts to all the $n$-grams in the corpus. Given a corpus $T$, the probability of each $n$-gram is given by:

$$P_\lambda\left(\mathbf{w}_k \mid w_{k-n+1}^{k-1}\right) = \frac{|w_{k-n+1}^k| + \lambda}{\sum_w \left(|w_{k-n+1}^{k-1}w| + \lambda\right)}$$

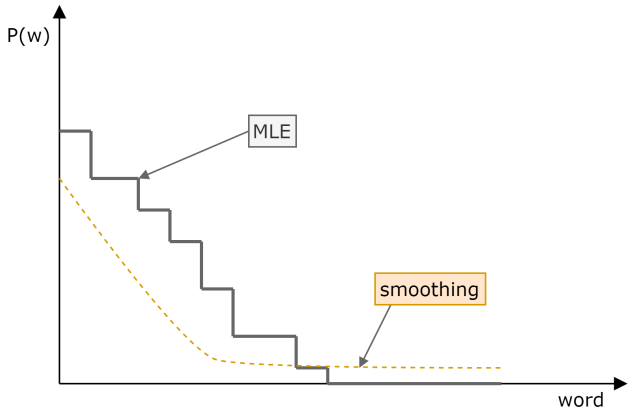



**Figure 2.1: Maximum likelihood estimation vs smoothing.**

Where $V$ is the vocabulary of the corpus. The parameter $\lambda$ can be adjusted. When $\lambda = 1$, we have the **add-one smoothing** that simply adds one to the count of all $n$-grams in the corpus.

### Good-Turing smoothing

**Good-Turing smoothing** consists of changing the frequencies of $n$-grams that appear $r$ times in a corpus $T$ to $r*$. With $r*$ computed as follows:

$$r* = (r+1)\frac{n_{r+1}}{n_r}$$

Where $n_r$ is the frequency of $n$-grams that appear $r$ times in $T$. In other words, we estimate the probabiliy of $n$-grams that appear $r$ times with the probability of $n$-grams that appear $r+1$ times.

### Back-off and interpolation model

The idea behind **back-off** and **interpolation smoothing** is that we combine the $n$-gram model with lower-order model(s).
Considering a trigram model, with back-off smoothing we use the count of the trigram (*if non-zero*), otherwise the bigram, otherwise the unigram. For example, **Katz smoothing** is a back-off model :

$$P_{katz}\left(\mathbf{w}_k | w_{k-n+1}^{k-1}\right) = \begin{cases} P^*\left(w_k | w_{k-n+1}^{k-1}\right) & \text{, if } |w_{k-n+1}^{k}| > 0 \\ \alpha\left(w_{k-n+1}^{k-1}\right) P_{katz}\left(w_k | w_{k-n+2}^{k-1}\right) & \text{, otherwise} \end{cases}$$

The parameter $\alpha$ allows to give more or less weights to the back-off values (*i.e., probabilities of the lower-order model*). $P^*$ is usually a smoothed distribution using **Good-Turing**.

In contrast to back-off models, interpolation models use systematically lower-order model(s). For example, given a trigram model, we have :

$$\begin{aligned} P^*\left(w_k \mid w_{k-1}\, w_{k-2}\right) = \lambda_1\, P\left(w_k \mid w_{k-1}\, w_{k-2}\right) + \\ \lambda_2\, P\left(w_k \mid w_{k-1}\right) + \\ \lambda_3\, P\left(w_k\right) \end{aligned}$$

Optimal values of the $\lambda$ parameters need to be learned using a validation corpus (*i.e., see next section about LM evaluation*).

Finally, another technique based on interpolation is **Kneser-Ney smoothing** [Kneser and Ney, 1995]. It is considered as one of the best smoothing technique and is widely used in the literature [Chen and Goodman, 1996; Goodman, 2001].

### 2.4.4 Language Model Evaluation

We distinguish two processes to evaluate a language model:

1. **Intrinsic evaluation**. We evaluate the quality of the model independently from any application.
   A good language model will predict with high probability the content of a new test document (*e.g., unseen during the training phase*). If we consider an English corpus consisting of text documents, a good model will have a low-level of **uncertainty** when predicting a word sequence in a previously unseen English document. The level of uncertainty of a language model can be measured by the **cross-entropy**. Given a $n$-gram language model $L$ and a word sequence $\mathbf{w}_1^n = w_1, w_2, ..., w_n$, the cross-entropy is computed as:
   $$H_L\left(\mathbf{w}_1^n\right) = -\frac{1}{n}\sum_{i=1}^{n} \log P(w_i \mid w_{i-n+1}^{i-1}).$$
   For the case of a $n$-gram model, the cross-entropy is the average number of bits required to predict the $n^{th}$ word given the $n-1$ previous words. Consequently, a model that has low entropy on some text documents has a low-level of uncertainty and predicts with confidence the content of the documents.

2. **Extrinsic evaluation**. The quality of the LM is evaluated on an application using proper evaluation metrics (*e.g., Precision/Recall, F-measures, Precision@k, Mean Reciprocal Rank, ...*). The application presented in Section **2.4.1** is an example of extrinsic task on which we can evaluate a language model. This is an example of a typical recommender system that provides lists of most probable suggestions and that can be evaluated using precision/recall curves or rank-sensitive metrics such as maximum reciprocal rank.

The extrinsic evaluation is the best way to evaluate a LM since it gives the performance of the model on a useful end-task. However, the intrinsic evaluation is useful to give an overview of the quality of the model and also to optimize the values of its hyperparameters (*e.g., smoothing parameters, n-gram model order, ...*) by minimizing the cross-entropy.

### 2.4.5 Application of $n$-gram LMs for Source Code Modeling

Recent works for modeling source code have focused on learning probabilistic models of code. Approaches based on $n$-gram language models have shown to be useful to learn regularities in code and to build code completion tools [Allamanis and Sutton, 2013b; Hellendoorn and Devanbu, 2017; Hindle et al., 2012; Nguyen et al., 2013; Rahman et al., 2019; Tu et al., 2014].

In this section, we discuss two examples from the literature of the usage of $n$-grams LMs for source code modeling and illustrate the usefulness of the intrinsic evaluation. In Chapter **3**, we discuss the usage of such models for code completion.

#### Naturalness of software

*Hindle et. al's* work on source code modeling was a breakthrough and has given rise to the hypothesis of the **naturalness of software** [Hindle et al., 2012]. This work has opened up a new area of research in source code modeling that is based on the utilization of NLP. The authors trained $n$-grams models on several projects to show the repetitive aspect of code. Figure **2.2** illustrates a comparison between the cross-entropy of 10 projects and the cross-entropy of an English corpus. The top curve shows the cross-entropy for the English corpus and the lower boxplots show the cross-entropy for the 10 projects. As we can see, the cross-entropy of the code is significantly lower and the authors conclude that code is more regular than English regardless of the programming language.

#### Naturalness of software revisited

A recent work extending the **naturalness of software** hypothesis has shown interesting results as discussed in Section **2.3** [Rahman et al., 2019]. The authors replicated *Hindle et al.*'s work and showed that when filtering syntax tokens, the cross-entropy of the code increases significantly. The conjunction of this result with further results in the paper has led to the conclusion that identifiers are the main responsible for the high-level of cross-entropy of code. Figures **2.3** depicts these observations. Therefore, one of the main challenge of source code modeling is to learn identifier representations that can help for an extrinsic task such as code completion.

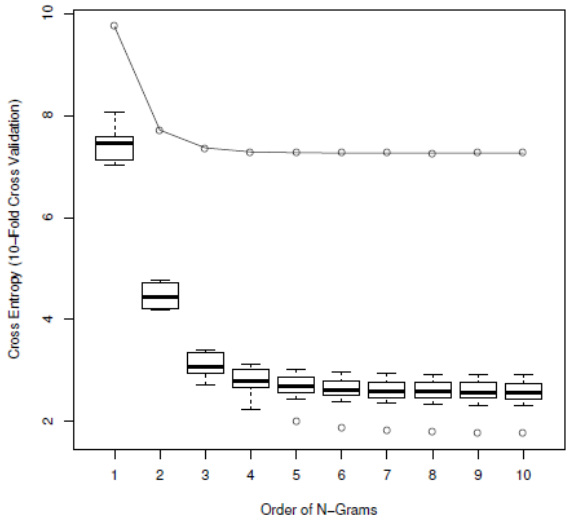
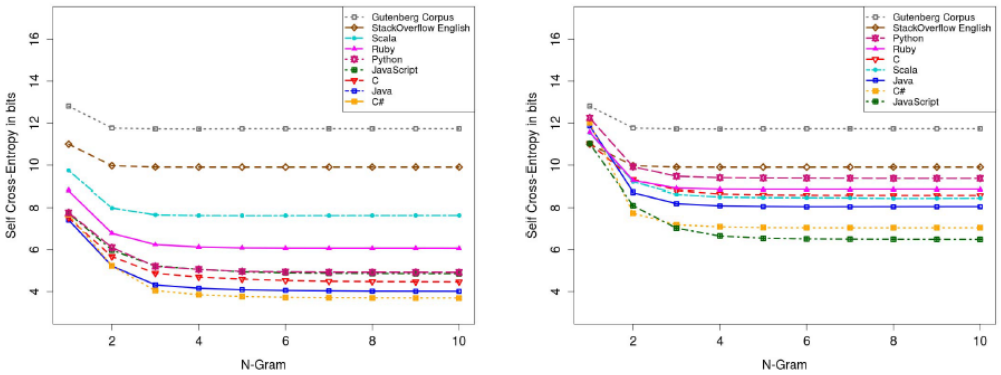



**Figure 2.2: Results of *Hindle et al.*'s experiments on source code modeling using $n$-grams LMs. Comparison of English Cross-Entropy versus the Code Cross Entropy of 10 projects [Hindle et al., 2012].**




(a) SelfCrossEntropy with SyntaxTokens.

(b) SelfCrossEntropy *without* SyntaxTokens.

**Figure 2.3: Results of *Rahman et al.*'s experiments on source code modeling using $n$-grams LMs. Comparison of Cross-Entropy without filtering syntax tokens (left) and with syntax token filtering (right) [Rahman et al., 2019].**

## 2.5 Distributional Approaches for Word Representation

The core of our approach is based on machine learning and word embedding. It is therefore essential to understand what are the inputs and outputs of such models and how the learning process works. This section first covers a general framework of machine learning. Then, it addresses some embedding-based approaches to learn vector representations of words and sequences.

### 2.5.1 Machine Learning

The objective of machine learning is to build mathematical and statistical models in order to automate a task by learning from sample data. A model is designed by a human and seeks to achieve one specific task. The model has parameters that are learned from the data. We distinguish two main learning approaches : (1) **supervised machine learning** which requires labelled data and allows classification or regression to be performed, (2) **unsupervised machine learning** that does not require the data to be labelled and generally focus on learning the underlying structure of the data (*e.g., density estimation, clustering*).

#### General Framework of Learning

We assume a dataset $D$ generated by an unknown process $D_n = (Z_1, Z_2, ..., Z_n)$ where the samples are *i.i.d* (*independent and identically distributed*) and drawn from the same unknown distribution $P(Z)$.

- In **supervised machine learning**, the samples are labelled $Z = (X, Y)$. where $X \in \mathbb{R}^d$ is a vector of $d$ features and $Y \in \mathbb{R}$ (*regression*) or $Y \in [1, ..., N]$ (*classification*) is the label.
  The objective of the training phase is to learn a predictor $f_\theta$ and find the set of parameters $\theta$ that minimizes a loss function:

$$\arg\min_{\theta} \frac{1}{T} \sum_{t} l(f(\mathbf{x}^{(t)}; \theta), y^{(t)})$$

  This framework is known as the **empirical risk minimization** (**ERM**). The loss function computes the difference between the real target $y$ of a sample and its predicted value $f(x)$. A typical loss function for regression is the quadratic error $L(f(x), y) = (f(x) - y)^2$. In classification, the loss function usually depends on the type of model that we are training (*e.g., cross-entropy in neural networks*).

  The parameters $\theta$ are learned using the **gradient descent algorithm**:

  – initialize parameters $\theta$ (*randomly*)

- for each training example $(x^{(t)}, y^{(t)})$
  - $\Delta = -\nabla_\theta l(f(\mathbf{x}^{(t)}; \theta), y^{(t)})$
  - $\theta \leftarrow \theta + \alpha \Delta$

The goal of this algorithm is to minimize the loss function by going in the direction of the slope of the gradient. In this example, we perform a **stochastic gradient descent** that consists of updating the parameters $\theta$ each time that the model has gone through one training sample. Another way to update the parameters is to perform a **(mini)-batch gradient descent** where the parameters are updated after that the model has gone through $n^*$ samples. In that case, the gradient is computed over a (mini)-batch of $n^* \leq n$ samples $\in Z_n$.

In conclusion, supervised machine learning aims at finding the best predictor $f_\theta$ that will generalize well on unseen samples. That is, doing well on predicting the labels of unseen data.

- In **unsupervised machine learning**, the samples are not labelled and are real vectors of $d$-dimensions $Z \in \mathbb{R}^d$. In this scheme, the objective may be multiple. For example, we can do clustering to find clusters of similar groups of samples. Or, we can estimate a density distribution $p(Z \mid \theta)$ that maximizes the likelihood of the data (*i.e., MLE*).
  In some cases unsupervised models are called **self-supervised**. The idea of such models is to generate automatically a prediction task within the learning process. In the next section, we discuss **Word2vec** which is a self-supervised neural network model.

### 2.5.2 Word Embedding

**Word embedding** is a technique commonly used in natural language processing to learn a mapping of words into an **high-dimensional vector space**. The objective is essentially to embed the meaning of words into vectors. The notion of word embedding is highly related to **distributional semantics**. That is quantifying some semantic similarities between words or concepts that appear frequently in the same context in a large corpus of textual data. Two words that have close vector representation are meant to be similar in meaning. For example, it is likely that *senate* and *politic* would be close in an embedding space. Before introducing word embedding, we explain **one-hot representation** which is the simplest way to represent words and is usually how we choose to represent words as input to any embedding-based neural networks.

### One-hot representation

**One-hot vector** represents a word as a $\mathbb{R}^{|V|\times 1}$ vector with 0's and one 1 at the index position of the word in the vocabulary $V$. For instance:

$$\text{senate} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \text{politic} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The main issue with one-hot representation is that there is no way to compute the similarity between both words because the vectors are orthogonal:

$$one\_hot(\text{senate})^T one\_hot(\text{politic}) = 0$$

Another issue is about the storage of such vector representations. In fact, in a real case scenario a vocabulary could contain thousands of words and one-hot vector representations would yield a huge $\mathbb{R}^{|V|\times|V|}$ sparse matrix.

Therefore, it is needed to reduce the size of the vector representations to a reasonable size and determine an embedding space that captures semantics of words. In the remainder of this section, we present several machine learning models that are able to learn meaningful word representations.

### Word2vec

One of the most-known framework for learning distributed[1] vector representation of words is **Word2vec** [Mikolov et al., 2013a,b]. *Mikolov et al.* proposed two neural network architectures that are able to learn vector representation of words from large corpus containing billions of words. The first architecture is called **Continuous Bag-of-Words** (**CBOW**). In this architecture, the model learns word representation that can best predict a center word given the surrounding words. The second architecture is called **Skip-Gram** (**SG**). Instead of predicting a center word given the surrounding words, the SG model aims to best predict the surrounding words given the center word. Figure **2.4** illustrates both architectures. One of the main advantage of Word2vec is that the model is simple and has a low computational cost compared to traditional neural network language models [Bengio et al., 2003; Mikolov et al., 2013a].

[1] We say distributed because the meaning of a word is distributed across the components of its vector.

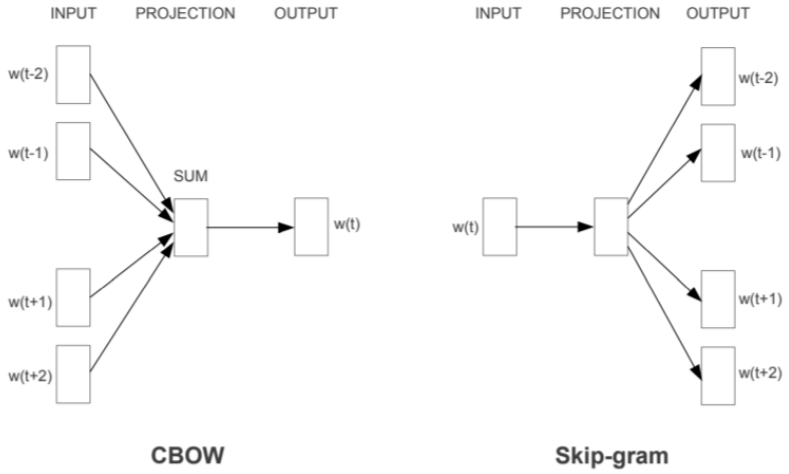



**Figure 2.4: Word2vec CBOW and SG architectures [Mikolov et al., 2013a].**

### Word2vec – Continuous Bag-of-Words Model

Word2vec is a single hidden-layer neural network. Figure **2.5** depicts the CBOW architecture in more details. The model takes as input a corpus of sentences. For each sentence, the model shifts over it to generate training samples $\big((x_1, x_2, ..., x_C), y_j\big)$. Each shift generates an input context $(x_1, x_2, ..., x_C)$ and a center word $y_j$ determined by a size of window (*e.g., a window size of 2 generates contexts with maximum 4 words surrounding the center word*). Regarding Figure **2.5**, we assume the following:

- $V$ = size of the vocabulary
- $\frac{C}{2}$ = window size
- $N$ = dimension of the embedding space (*i.e., hidden layer*)

The inputs $x_{Ck}$ are one-hot vectors of the context words. $\mathbf{W} \in \mathbb{R}^{V \times N}$ is the input matrix where each row is a $N$-dimensional vector representation $\mathbf{v}_w$ of a word $w$ in the vocabulary. $\mathbf{W}' \in \mathbb{R}^{N \times V}$ is the output matrix where each column is a $N$-dimensional vector representation $\mathbf{v}'_w$ of a word $w$. Thus, each word has two vector representations:

- $\mathbf{v}_w$ is the vector representation of $w$ when it is in the input context.
- $\mathbf{v}'_w$ is the vector representation of $w$ when it is the center word to be predicted.

These vectors are the parameters $\theta$ of the model and need to be learned in order to make the model good at predicting the center word.

The hidden layer is obtained by averaging the context vectors :

(*lookup in* $\mathbf{W}$) $$\mathbf{v}_c = \mathbf{W}^T x_c$$

(*average*) $$\mathbf{h} = \frac{\mathbf{v}_1 + \mathbf{v}_2 + ... + \mathbf{v}_c}{C}$$

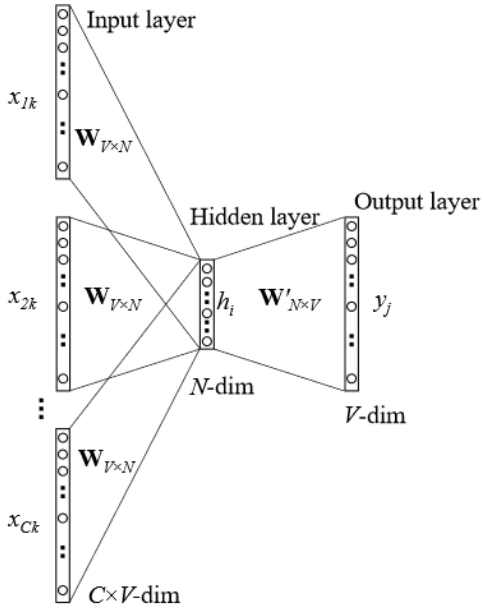



**Figure 2.5: Word2vec - CBOW architecture [Rong, 2014].**

Then, we generate a score for each word in the vocabulary using the output matrix $\mathbf{W}' \in \mathbb{R}^{N\times V}$ :

$$u_j = (\mathbf{v}'_{w_j})^T \mathbf{h} \tag{2.2}$$

Where $\mathbf{v}'_{w_j}$ is the $j$-th column $\mathbf{W}'$. The dot product in equation **2.2** gives a high value if the average of the context words' embeddings ($\mathbf{h}$) is close to the embedding of the word $w_j$. The matrices $\mathbf{W}$ and $\mathbf{W}'$ are learned in order to maximize the probability of this score, thus this equation gives us the intuition that close words are more likely to have similar vectors.
Then we turn the scores into probabilities using **softmax**:

$$P(w_j \mid w_1, ..., w_C) = \widetilde{y}_j = \frac{\exp(u_j)}{\sum_{i=1}^{V} \exp(u_i)} \tag{2.3}$$

The goal is that $\widetilde{y}_j$ matches the true probability of the word $w_j$ given by its one-hot vector representation $y_j$. For this, the model has to learn the weights of the matrices $\mathbf{W}$ and $\mathbf{W}'$. As we have seen in Section **2.5.1**, we need to define a loss function that the neural network will minimizes using gradient descent. A common choice is the cross-entropy which allows to compare probability distributions:

$$H(\widetilde{y}, y) = -y_j \log(\widetilde{y}_j)$$

Thus, the closer $\widetilde{y}_j$ is to $y_j$, the closer the cross-entropy $H(\widetilde{y}, y)$ will be to 0.

The objective function is to maximize the probability to observe the actual current word given the surrounding words (eq. **2.3**) :

$$\begin{aligned}\max P(w_j \mid w1, ..., w_c) &= \max \tilde{y}_j \\ &= \max \log \tilde{y}_j \\ &= \max \log \frac{\exp(u_j)}{\sum_{i=1}^{V} \exp(u_i)} \\ &= u_j - \log \sum_{i=1}^{V} \exp(u_i)\end{aligned}$$

This is equivalent to minimizing $-\log \tilde{y}_j$, the negative log-likelihood which is the cross-entropy. The weights $\mathbf{W}$ and $\mathbf{W}'$ are updated by computing the derivative of the cross-entropy *w.r.t* to matrices $\mathbf{W}$ and $\mathbf{W}'$ using the update rule discussed in Section **2.5.1**.

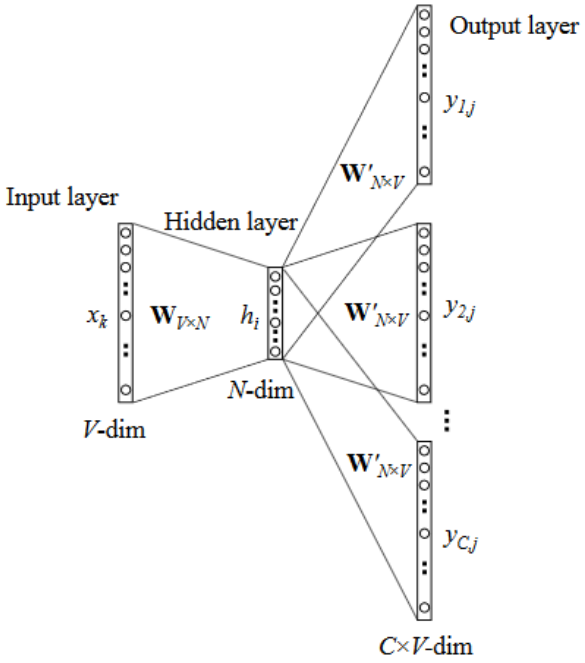



**Figure 2.6: Word2vec - SG architecture [Rong, 2014].**

### Word2vec – Skip-Gram Model

As opposed to the CBOW architecture, in SG a unique context word is used to predict the surrounding words. The formulation of the input and output matrices $\mathbf{W}$ and $\mathbf{W}'$ is identical to CBOW. The difference is that the model computes $C$ multinomial distributions with softmax in the output layer (*i.e., one for each output word*) and assumes complete

independence between the surrounding words (*i.e., Naive Bayes assumption*). Figure **2.6** shows the SG architecture in details[2].

#### GloVe

One of the main drawback of Word2vec is that the model does not make use of the global statistics of the input corpus. In contrast, **Global Vector for Word Representation** (**GloVe**) leverages the global co-occurrence statistics of words in the training corpus in order to produce finest word embedding [Pennington et al., 2014]. The authors showed that their approach outperforms Word2vec on the analogy task given the same model configuration[3].

#### FastText

The above mentioned embedding models suffer from a lack of generalization for words not seen during the training. **FastText** aims to tackle this problematic by exploiting the morphological structure of words [Bojanowski et al., 2016]. In fact, this model allows to learn embeddings at the character level. The model cuts words into characters $n$-grams (*or subwords*) as follows :

$$< \text{where} >:=< \text{wh, whe, her, ere, re} > \text{and} < \text{where} >$$

The word itself is included to learn representations of the full words along with its sub-words. The advantage of this approach is that the embedding of word that has not been seen during the training can be reconstructed using the character $n$-grams embeddings.

#### Contextual Word Embedding Models

Recent works have focused on learning contextual word embedding [Devlin et al., 2018; Peters et al., 2018]. These models allows to learn embeddings of words by considering the full context of a word. In that scheme, a word can have several vector representations depending on the context in which it appears. In that way, the model is able to efficiently understand the intent behind a sentence.
**Bidirectional Encoder Representations from Transformers** (**BERT**) [4] developed by Google is considered as the state-of-the-art language model in a lot of NLP tasks. Recently, Google announced that they are applying BERT to enhance the efficiency of their search engine [5].

[2] Details about the implementation of *SG* and optimization of both models with negative sampling and hierarchical softmax can be found in the following paper [Rong, 2014]

[3] Details about the model can be found in the original paper and at the following page: http://web.stanford.edu/class/cs224n/readings/cs224n-2019-notes02-wordvecs2.pdf

[4] The following site explains the Transformer model : http://jalammar.github.io/illustrated-transformer/

[5] https://www.blog.google/products/search/search-language-understanding-bert/

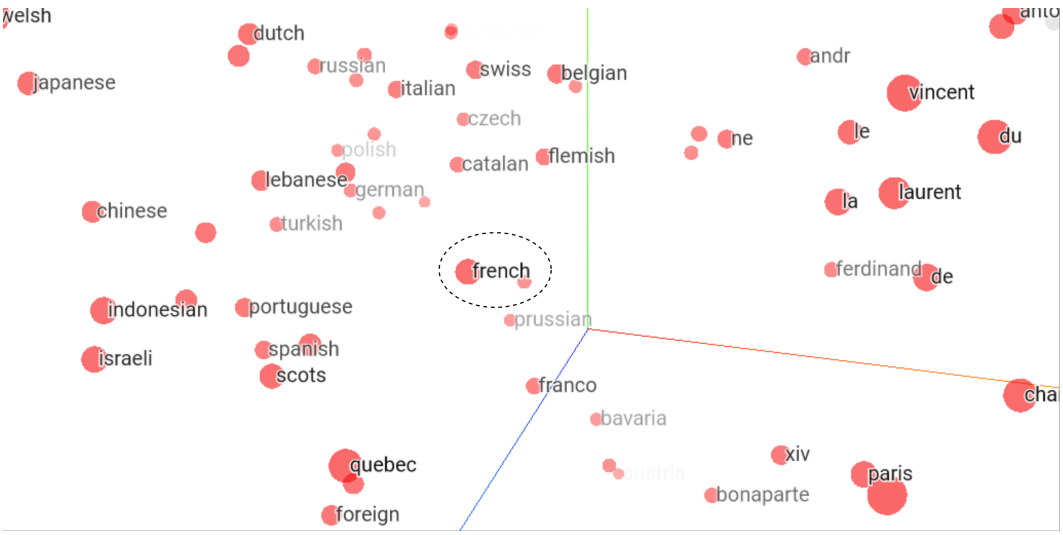



**Figure 2.7: Visualization of Word2vec embeddings using t-SNE.**

### Word embedding – Linguistic Regularities

In a similar way to language models, word embedding models can be evaluated intrinsically or on an extrinsic task. The most common ways to perform the intrinsic evaluation are the relatedness and analogy tasks [Schnabel et al., 2015]. In the former, we establish scores between pairs of words and we check whether two related words have close embeddings using a distance metric such as cosine similarity. In the latter, the goal is to find pairs of words that are analogously close.

The analogy task has been made popular by *Mikolov et al.* The authors showed that Word2vec is capable of capturing syntactical and semantic regularities that allows to solve analogy equations [Mikolov et al., 2013c]. The most known example of analogy is probably the following : $vec(\text{King}) - vec(\text{Man}) + vec(\text{Woman}) \simeq vec(\text{Queen})$.

On the other hand, the relatedness task can be achieved by reducing the dimension of the embeddings and visualizing the reduced vectors on a 2-$D$ or 3-$D$ space. The dimensionality reduction can be made using **t-SNE** or **PCA**, for example. Figure **2.7** shows a 3-$D$ visualization of Word2vec embeddings trained on Wikipedia[6]. We focus on the word "*french*" and as we can see the closest words seem to be highly related (*e.g., english, france, spanish, german and dutch are the top-5 most similar words*).

These analysis can be extended to any word embedding model. The intrinsic evaluation gives us an idea of the goodness of the word vectors. Ideally, we want the intrinsic evaluation to be positively correlated to the extrinsic task.

[6]https://projector.tensorflow.org/

### 2.5.3 Sentence Embedding

Word embedding models are capable of learning embeddings at the word level. Given a training corpus as input, a word embedding model learns a vector representation for each word in the vocabulary. One of the drawback of this model is that there is no inherent scheme to the model to learn embedding of sequence of words. Such an approach would, for example, allow us to compute the similarity between two text documents (*e.g., of variable size*). It would be possible to determine vector representations of sentences with a word embedding model by averaging the word vectors of the sentences, but it has been shown to be not efficient. Sentence embedding models aim to tackle this problematic by learning vector representation of variable-length texts.

#### Paragraph Vector Model

The **paragraph vector** (**PV**) model (or **Doc2vec**) is an extension of Word2vec proposed by *Le and Mikolov* [Le and Mikolov, 2014]. PV models learn vector representations (*paragraph vectors*) of sequences of textual data of variable size (*document, phrases, news article...*). In this model, each input sequence has a unique corresponding paragraph vector that is learned along with the word vectors. Paragraph vectors are not just concatenation and average of word vectors contained within the paragraph. Instead, paragraph vectors are asked to contribute to a predictive task as for words in *Word2vec*. There exist two architectures for the learning process : **distributed memory** (**PV-DM**) and **distributed bag-of-words** (**PV-DBOW**).

- In PV-DM, the model randomly sample contexts within the paragraph. The contexts are determined by a window size. Then, an average or concatenation of the paragrah vector and the word embedding is used to predict the last word of the sampled context. Figure **2.8** illustrates this architecture.

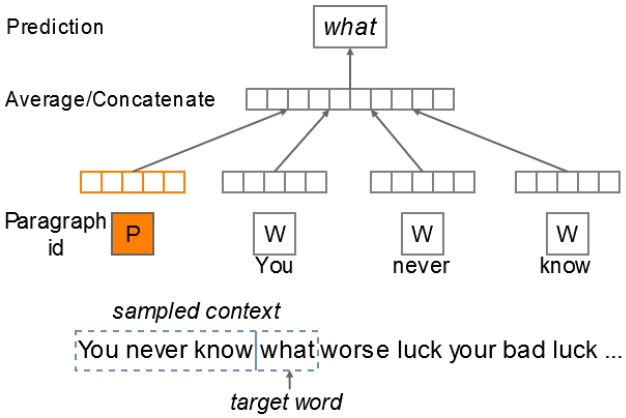



**Figure 2.8: PV-DM. The context is randomly sampled using a window size of 4. Words and paragraph embeddings are averaged or concatenated to predict the target word.**

- In PV-DBOW, the model sample contexts similarly to PV-DM. However, context words are ignored in the input. The paragraph vector is asked to predict random words from the sampled context. This is illustrated in Figure **2.9**.

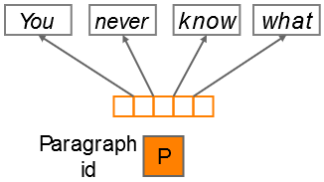



**Figure 2.9: PV-DBOW. The context is ignored in the input. The paragraph vector is used to predict the context words.**

The advantage of the PV model over Word2vec is that the model is able to learn representation of variable-length texts. As a result of the learning phase, the paragraph vectors can capture semantic properties about a whole sequence of textual data. This model has shown to be useful in topic modeling and several NLP tasks [Dai et al., 2015; Hashimoto et al., 2016; Lau and Baldwin, 2016].

## 2.6 Conclusion

**Conclusion**

In this chapter we have gone through common techniques used in natural language processing and have shown their usage on practical examples and on source code. These techniques underlie to some degree approaches involving language models or embedding-based models for source code modeling.

Then, we have discussed $n$-gram language models that allows estimating words and sequences probabilities in a corpus. In addition to their use for a variety of tasks, we have shown that an inexpensive intrinsic evaluation of these models enables to determine the quality of the models and the predictability of a corpus. We have demonstrated the good practice of this type of evaluation on examples taken from the literature on source code modeling. These examples have highlighted the non-trivial aspect of source code modeling and the need for more in-depth approaches.

Finally, we have addressed word and sentence embedding whose use have become standard in many NLP tasks. These models are at the core of our approach and in Chapter **4**, we show how we leverage their use in order to build a code completion system and to enhance the effectiveness of a well-known code completion tool.

# CHAPTER 3

## RESEARCH ON CODE COMPLETION

Neural approaches, *n*-gram and embedding-based language models have been widely used for automating tasks of the software development lifecycle [Allamanis et al., 2017; Chen and Monperrus, 2019]. However, we focus on code completion which has been a very active field over the past few years and is related to our work. We differentiate between three types of approach. Firstly, there are traditional approaches that essentially extract features from the project under development to provide suggestions. Then, *n*-gram-based methods based on the naturalness of software hypothesis. Finally, neural-based approaches that leverage structured representations of code such as ASTs.

## 3.1 Traditional Approaches

The first learning-based approach for code completion were proposed by *Bruch et al.* [Bruch et al., 2009]. In their work, the authors compared three systems : (1) a frequency-based system that determines the relevance of a call *w.r.t.* its frequency in the training set, (2) an association rule method that attempts to associate relevant calls with call site contextual data, and (3) a k-nearest-neighbor (**kNN**) approach that provides recommendations by matching the development context with code snippet examples. Later on, Proksch et al. extended this work by using Bayesian networks and gathering more context information [Proksch et al., 2015].

Recently, *Nguyen et al.* exploited typical recommender system techniques for APIs recommendation [Nguyen et al., 2019]. They developed FOCUS, a context-aware collaborative filtering APIs recommender. The system allows to suggest relevant API calls by matching

the call site context with contexts in sample projects that are close to the one under development.

Even though these approaches have shown to be efficient for APIs recommendation, they are not flawless. In fact, the main issue of these techniques is that they rely on manually extracted features from the code and are designed for particular APIs. Moreover, collaborative filtering techniques require ever-increasing computational costs as the size of the training set increases.

## 3.2 *n*-gram-based Approaches

With the hypothesis of **naturalness of software** (see Section **2.4.5**), *Hindle et al.* outlined the possibility to use *n*-gram language models for code completion by predicting call sites given the previous code tokens [Hindle et al., 2012]. The authors showed that a simple trigram model is able to provide relatively accurate suggestions and increases the correctness of Eclipse Suggestion Plug-in. In contrast to traditional approaches, this kind of method does not require hand-coded features because it is only based on the sequential aspect of code.

*Tu et al.* extend this work and showed that the code is locally repetitive [Tu et al., 2014]. That is, some repeated *n*-grams are generally close in a code artifact. By adding a cache component to the *n*-gram language model, the authors have shown the model to be more efficient than the cacheless model in term of correctness of the code suggestions. The cache is made of local patterns captured in the code (*e.g., occurring in the file being developed*).

*Hellendoorn and Devanbu* extended this approach by improving the cache component with information about the scope of the call site such as the module in which the developer is coding a specific method [Hellendoorn and Devanbu, 2017]. The authors showed that a fine-tuned cache *n*-gram model is able to outperform neural approaches. Similarly, *Raychev et al.* compared the performance of *n*-gram and neural language models for Android API code suggestion [Raychev et al., 2014]. Their results show that a trigram language model is as good as a recurrent neural network (**RNN**).

These last few works take advantage of the sequential nature of the code to perform code completion without filtering predictable tokens, such as syntax tokens. As discussed in Section **2.4.5**, it has been shown that these tokens make the code artificially predictable [Rahman et al., 2019]. Therefore, even though these experiments show generally good results, they might be overestimated for realistic scenarios where the syntax tokens can be

completed using type-based completion tools.

## 3.3 AST-based and Neural Approaches

*Nguyen et al.* were the first to use AST-based language model to learn higher-level patterns than $n$-gram language models to improve API code suggestion [Nguyen and Nguyen, 2015].
Recent approaches using deep learning have mainly focused on learning representations of AST with long-short term memory (**LSTM**), attention-based neural networks, and more recently transformer models [Hochreiter and Schmidhuber, 1997; Vaswani et al., 2017].

*Bhoopchand et al.* use pointer networks model[1] to learn long-range dependencies in Python ASTs for identifiers completion [Bhoopchand et al., 2016]. *Li et al.* use the same approach with a focus on out-of-vocabulary identifiers [Li et al., 2017]. *Karampatsis et al.* proposed a LSTM neural networks that is able to suggest out-of-vocabulary identifiers by learning the internal structure of code tokens (*e.g., subtokens*) [Karampatsis et al., 2020].

*Svyatkovskiy et al.* compare several neural network architectures for method and API recommendations in Python [Svyatkovskiy et al., 2019]. They learn AST-based representations of code snippets to perform the completion by comparing a call site context with the AST representations learned by the model. In a subsequent paper, *Svyatkovskiy et al.* define a framework using the same approach combined with existing code completion tools [Svyatkovskiy et al., 2020]. The authors define a re-ranking task of code recommenders in order to only recommend tokens that are type compliant.

A few recent works on code completion have focused on feeding AST trees into state-of-the-art deep neural language models in order to predict any kind of code token. *Alon et al.* proposed an approach where a transformer model learns to predict an AST node (*i.e. a code token*) given all possible AST paths leading to this node [Alon et al., 2019]. Their approach also allows the prediction of out-of-vocabulary token. *Kim et al.* designed the same kind of approach but compared several ways to feed AST trees into a transformer model [Kim et al., 2020] and focused the evaluation of their model for predicting specific types of tokens (*e.g., attribute access, variable name, ...*).
These works reported lower effectiveness than previous works on APIs and identifiers completion due to their broader application scope and are less related to our approach where we focus on function-call completion.

[1] a model derived from Bahdanau et al's attention mechanism [Bahdanau et al., 2014]

## 3.4 Conclusion

**Conclusion**

Previous works on code completion cover a broad range of techniques that have proven to be effective to some extend. Traditional approaches suffer from high computational costs that can eventually lead to unwanted latency when generating recommendations. $n$-gram-based and neural approaches generally do not filter syntax tokens or are designed to predict specific APIs which ease the completion task.

Henceforth, there is room for improvement, especially for building code completion engines that are not specific for APIs and that have low-computational costs. Consequently, our attempt to train embedding-based model on function identifiers for function-call completion is relevant and complementary to the existing literature.

# Part II

# Contribution

# CHAPTER 4

# CODE COMPLETION IN THE TIME OF MASSIVE DATA

## 4.1 Introduction

This chapter provides a detailed discussion of our contribution. The first section presents our approach for building a code completion system based on embedding models and explains its usage for direct completion or integration within an existing typing-based tool. Then, we present the research questions, an analysis of our data source and the metrics used for evaluating our systems. Finally, we analyze our results, address some threats to the validity of our experiments and draw a conclusion.

## 4.2 Approach

To illustrate the rationale behind our approach, let us consider the situation in which, Ulwazi, a Java developer, is writing the method :

```
1 // ...
2 public long size() throws IOException {
3   if (!file.isFile()) {
4     throw new FileNotFoundException(file.toString());
5   }
6   return file.? // prediction (ctrl+space)
7 }
```

**Listing 4.1: Motivating example**

Consider also that Ulwazi is coding in an IDE that incorporates, among other features, a code completion plug-in such as Eclipse content assist[1] that suggests function calls.

[1] https://www.eclipse.org/documentation/

In line 6, after she types " . " the plug-in is invoked, and the latter will provide a suggestion list of possible items including function calls that could follow "*file.*". The plug-in exploits static environment information about the currently opened code artefact (*e.g., imports, language typing...*). The produced suggestion list is generally exhaustive, often long, and ordered alphabetically. Thus, it is more likely that the correct suggestion will not appear at the top of the list, and developers like Ulwazi will waste a valuable time browsing through the list.

Therefore, our objective is to alleviate the burden of developers by providing completion suggestion lists that are: (1) of limited size, and (2) ordered by pertinence so that the correct suggestion is likely to appear in the top positions.

Our approach is based on the hypothesis that there exist recurring function-call patterns in large corpus of source code. Those patterns embody some semantics about high-level concepts, which may appear in different programs with slight linguistic variations. For instance, coming back to the example of the method "*size*" that computes the size of a file, the first step consists in checking whether the input is a file, by calling, for example, a function "*.isFile()*". If it is not, one may want to raise an exception with a representation of the file by calling "*.toString()*". The final step is to call a function "*.length()*" that outputs the size of the file. Our approach makes the assumption that such sequences of function calls are totally or partially recurrent among a lot of projects and that they capture most of the semantics of some higher-level concepts (in our case, "*get the size of a file*"). By comparing the previous function call sequence (*including the method name*) "*size, isFile, toString*" with function call sequences abstracted by an embedding-based machine learning model, it would be possible to determine that "*length*" is the most probable function call that comes after "*file.*".

In this work, we propose an approach to learn those high-level concepts and their relationships by training an embeddings model (*e.g., paragraph vector model*) on a big corpus of code. Once the model trained, we can take advantage of it for the code completion task in two ways: (1) building a suggestion list from scratch or (2) reordering the suggestions made by Eclipse content assist plug-in. We describe the learning process in Section **4.2.1** and the code completion in Sections **4.2.2** and **4.2.3**. Figure **4.1** illustrates both processes (*see also Appendix **A.1***).

### 4.2.1 Learning Concepts from Code

Word embedding models were originally designed for natural languages. They take as input a corpus of text and output vector representations of words and/or sequences of words (*e.g., with paragraph vectors*) as a result of the training process. Therefore, to

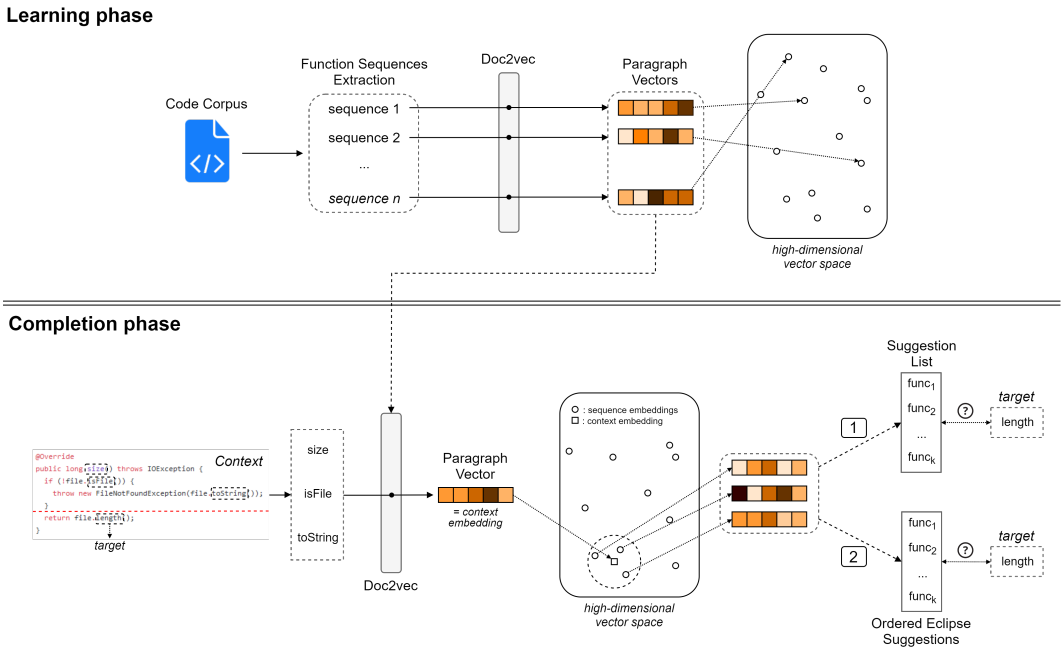



**Figure 4.1: Approach – General framework.**

use that kind of model with a programming language, we need to extract useful textual information from the code.

As discussed in the previous section, sequences of function calls embody a great part of the semantics of code. Therefore, it might be a good way to use these sequences as textual representations of the source code. Conversely, syntactical tokens such as *if, else, for* or a *parenthesis* carry less domain-specific information and considering them could lead to introducing a lot of noise in the learning, especially in the context of function call completion. It is also important to define how do we cut the code in order to produce sequences of functions and learn the paragraph vectors. We propose to limit the scope of a sequence to a method declaration and its body as for the *size* method. A method can be seen as a paragraph that is designed to deal with a particular concern, as we would do in a text. And, it is more likely that functions sequences within a small scope are more recurring than in a broader scope, *e.g.,* a whole class. Furthermore, limiting the sequences to a relatively small scope allows the model to learn specific and precise concepts. Our approach is not limited to one specific programming language, any corpus in any programming language can be used.

Our approach is also not limited to using sequences of functions. One alternative would be to consider subtokens of functions instead of the full functions names. Given a function name, the subtokens are words contained within it. For example, if we have a function that is called "*convertDateToString*", we tokenize the camel case and the resulting subtokens are "*convert, date, to, string*". This approach has been used in some previous

works [Allamanis et al., 2015, 2016; Karampatsis et al., 2020]. It has shown to be useful to summarize code snippets and for suggesting out-of-vocabulary method declaration names.

The learning process is described in the upper part of Figure **4.1**. The first step extracts function sequences from a corpus of code. Then, the sequences are used as input of a paragraph vector model (*Doc2vec*). Finally, the model learns high-dimensional vector representations of the function sequences (*paragraph vectors*).

### 4.2.2 Building Suggestion List

In this section, we explain how we can use the paragraph vector model to build a function call completion system from scratch as depicted in the lower part of Figure **4.1**. For that, we consider again the scenario of Section **4.2** in which Ulwazi is implementing a class method. The completion process is designed in four steps :

1. **Extraction of the context**. The context is made of the name of the method under development followed by the sequence of calls, in the method body, preceding the call site that triggers the completion.

2. **Inferring a paragraph vector**. Using the previously trained model, we infer a vector representation of the context (*context embedding*).

3. **Retrieving the most similar sequences to the context**. We use the embedding of the context to retrieve the closest paragraph vectors in the model. This can be done by finding the paragraph vectors that have the greatest cosine similarity to the context vector. Since paragraph vectors correspond to sequences of functions, this is like retrieving the most similar sequences of functions to the context. In our example, we find that three paragraph vectors are similar to the context vector and we retrieve them according to their closeness with the context vector (*ranked by decreasing cosine similarity*).

4. **Building the suggestion list**. We build a suggestion list, either of a fixed-length or depending on the similarity scores, using the retrieved sequences of functions. We iterate over these sequences in decreasing order of similarity. We add functions of the retrieved sequences to the suggestion list and stop when the list has reached its maximum allowed size or a threshold of the similarity score. This step corresponds to the branch **1** in Figure **4.1**.

### 4.2.3 Reordering Eclipse's Suggestions

Building the suggestion list from scratch does not guarantee that the suggestions are feasible in the current project. Instead of building a list of suggestions from scratch, we

can reorder the suggestions made by the Eclipse completion plug-in. This is shown in branch **2** in Figure **4.1**. The advantage of this approach is that Eclipse's plug-in provides suggestions depending on the packages/libraries imported in the code file in which the method is implemented. Therefore, the reordered suggestion list will only contain tangible function calls.

The process of producing the suggestion list is the same as above, except for the fourth step. In step four, we iterate over the most similar sequences to the context. For each function in a sequence, if the function appears in Eclipse's suggestion list then we add it to the final suggestion list. We use the same stopping criteria, *i.e.,* fixed length or threshold score. The reordering of Eclipse's list allow to suggest only function calls that are most likely to occur after the provided context and are correct *w.r.t.* typing and imports.

## 4.3 Evaluation Setup

Previous works have shown that source code is (*locally*) repetitive and predictable using statistical language models [Hindle et al., 2012; Tu et al., 2014]. Recent works have found that variable and function identifiers are the main responsible for the high-level of entropy of code and that syntax tokens artificially increase the source code predictability [Rahman et al., 2019]. Thus, one of the key challenges of learning high-level concepts from codes using sequences of functions lies in the high-level of unpredictability of those sequences. This leads us to address the following research questions:

- **RQ1 [Replication]:** ***How repetitive and predictable are function sequences in source code?***
  We reproduce previous works on naturalness of software [Hindle et al., 2012; Rahman et al., 2019]. We check whether our datasets satisfy the naturalness hypothesis introduced by Hindle et al. [Hindle et al., 2012]. Then, we ensure that our datasets have a level of cross-entropy in the same order of magnitude than in Rahman et al's experiments [Rahman et al., 2019]. To estimate $n$-gram language models we use kenLM [Heafield, 2011], a library that provides fast estimation and manipulation of $n$-grams models. As a first step, we estimate $n$-gram language models using our training sets for $n \in [2, 10]$. Then, we compute the cross-entropy on our test set.

- **RQ2:** ***Are paragraph vector embedding models capable of capturing concepts from the code?***
  We evaluate how well the paragraph vector model captures concepts by performing relatedness tests on some functions in the vocabulary of the trained model. Additionally, we reduce the dimension of the embeddings using t-SNE in order to visualize them in a 2-D space. The idea is to check whether close functions in the

model have consistent embeddings with respect to the semantics of the functions and their usage in the code.

- **RQ3:** ***Using the paragraph vector model, can we accurately suggest function invocations given a context?***
  We use the approach defined in Section **4.2.2**. The effectiveness of paragraph vector models is evaluated for a function call completion task. We use contexts of variable-length sampled from methods of our test projects to produce suggestion lists. We compute metrics defined in Section **4.3.3** based on the capacity of the model to build suggestion lists that contain relevant function calls.

- **RQ4:** ***Can we use the paragraph vector model in order to improve the suggestion ranking made by Eclipse's content assist plug-in?***
  We reorder Eclipse's suggestions using the process defined in Section **4.2.3**. We conjecture that the results will be better than those of *RQ3* since we take advantage of existing suggestion lists of Eclipse's plug-in. As for *RQ3*, we evaluate our approach using metrics defined in Section **4.3.3**.

### 4.3.1 Data Source

We use the GitHub Java Corpus [Allamanis and Sutton, 2013a] consisting of more than 14,000 open-source java projects collected from Github. The corpus' statistics are presented in Table **4.1**.

Before forming the training sets, we removed 20 projects from the original corpus to build a test set. We select these projects based on their high popularity in Github and to cover a broad range of application domains as shown in Table **4.2**. We also considered the diversity in size. Table **4.3** shows statistics for each test project, *i.e.*, the number of methods declared in each test project, the total number of call sites in these methods, and the percentage of function vocabulary that appear in the training dataset. We use the whole 20 projects to answer RQ1 and limit ourselves to the 10 projects in bold to answer the remaining questions. These 10 projects allow us to test the completion for more than 160.000 call sites.

| # Projects | LOC | Tokens |
|---|---|---|
| 14.785 | 352.312.696 | 1.501.614.836 |

**Table 4.1: GitHub Java Corpus statistics [Allamanis and Sutton, 2013a].**

| Name | Files | Description |
|---|---|---|
| aws-sdk-java | 65.204 | AWS SDK for Java |
| hibernate-orm | 10.088 | Object-relational mapping tool |
| gradle | 8.993 | Automation framework |
| spring-framework | 7.223 | Spring Framework |
| jclouds | 5.753 | Apps for cloud |
| hadoop-common | 5.685 | Apache Hadoop common |
| neo4j | 5.455 | Graph Database |
| druid | 4.069 | Database connection pools |
| spring-security | 2.700 | Spring security services |
| cassandra | 2.690 | Apache Cassandra |
| netty | 2.553 | Asynchronous network framework |
| mongo-java-driver | 1.586 | Java driver for MongoDB |
| antlr4 | 696 | Tool for Language Recognition) |
| junit | 468 | Testing framework |
| facebook-android-sdk | 458 | Facebook Android SDK |
| twitter4j | 446 | Twitter API |
| hystrix | 411 | Dependency managing tool |
| clojure | 176 | Clojure programming language |
| android-async-http | 84 | Asynchronous HTTP in Android |
| game-of-life | 30 | Game |

**Table 4.2: Test projects descriptions ordered by size (*number of files*).**

| Name | # Method Decl | # Function Calls | Coverage |
|---|---|---|---|
| aws-sdk-java | 245.430 | 1.799.530 | 76% |
| hibernate-orm | 30.867 | 278.124 | 86% |
| gradle | 26.913 | 120.123 | 84% |
| spring-framework | 44.433 | 332.121 | 88% |
| jclouds | 24.746 | 196.070 | 85% |
| hadoop-common | 46.449 | 347.093 | 88% |
| neo4j | 33.939 | 230.914 | 80% |
| druid | 15.674 | 123.341 | 87% |
| spring-security | 13.750 | 96.950 | 84% |
| cassandra | 23.398 | 188.773 | 83% |
| **netty** | 14.326 | 72.754 | 82% |
| **mongo-java-driver** | 7573 | 35.836 | 84% |
| **antlr4** | 2222 | 11.053 | 84% |
| **junit** | 2522 | 8144 | 94% |
| **facebook-android-sdk** | 1453 | 5689 | 80% |
| **twitter4j** | 2323 | 13.365 | 99% |
| **hystrix** | 1090 | 5790 | 78% |
| **clojure** | 1966 | 13.020 | 94% |
| **android-async-http** | 198 | 675 | 90% |
| **game-of-life** | 37 | 128 | 64% |

**Table 4.3: Test projects used in the experiments. Coverage is the percentage of functions that appear in the training set.**

| | # Function sequences | Tokens | Types |
|---|---|---|---|
| *no min count* | 10.702.667 | 86.219.928 | 3.141.457 |
| *min count (20)* | 10.702.667 | 74.820.025 | 222.730 |

**Table 4.4: Training set 1 (*full functions*). Statistics with and without minimum count parameter. Tokens corresponds to the number of method declarations and function calls in the dataset. Types is the number of unique tokens.**

For the training of the *n*-gram models and the paragraph vector model, we extract more than 10 millions function sequences from the filtered corpus. Figure **4.2** below depicts the distribution of the methods having a size between 2 and 20 (*i.e., method declaration + method function calls*). This graph shows that approximately 50% of the function sequences are of size 2 and 3 which constraints our model to learn concepts within small contexts. In Table **4.4**, we specify the number of tokens and word types with and without a minimum count parameter. This parameter is used with paragraph vector model to ignore functions that occur less than a specified threshold (*in the experiments, we keep functions that appear at least 20 times in the corpus*). The ignored functions are replaced by a common token "*UNK*". We can observe that when using this minimum count parameter, the number of types decreases drastically (*around 7% of types are kept*), but the total number of tokens does not decrease that much. This means that there is a significant amount of types that are not frequent among all projects and considering them in the learning phase could lead to learning a lot of noise.
Figure **4.3** corroborates with this result by showing that the top-20 method declarations/-function calls that occur the most represent ≈20% of the training corpus. Interestingly, this observation is in line with the Zipf's law that can be observed empirically in natural language corpus.

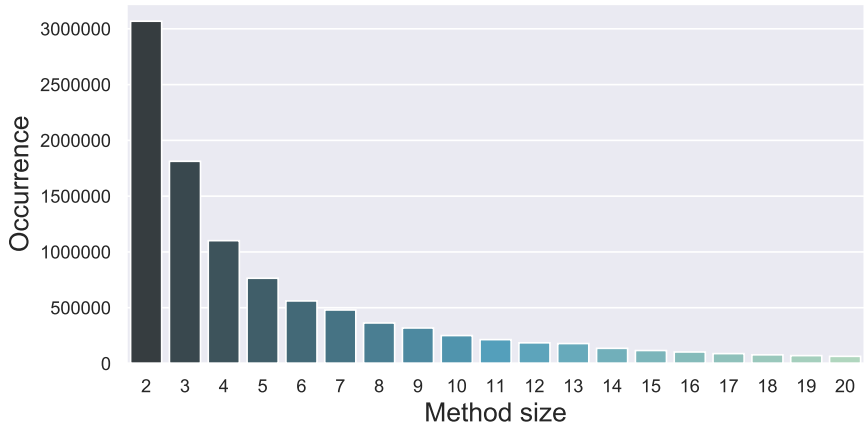



**Figure 4.2: Distribution of the method sizes.**

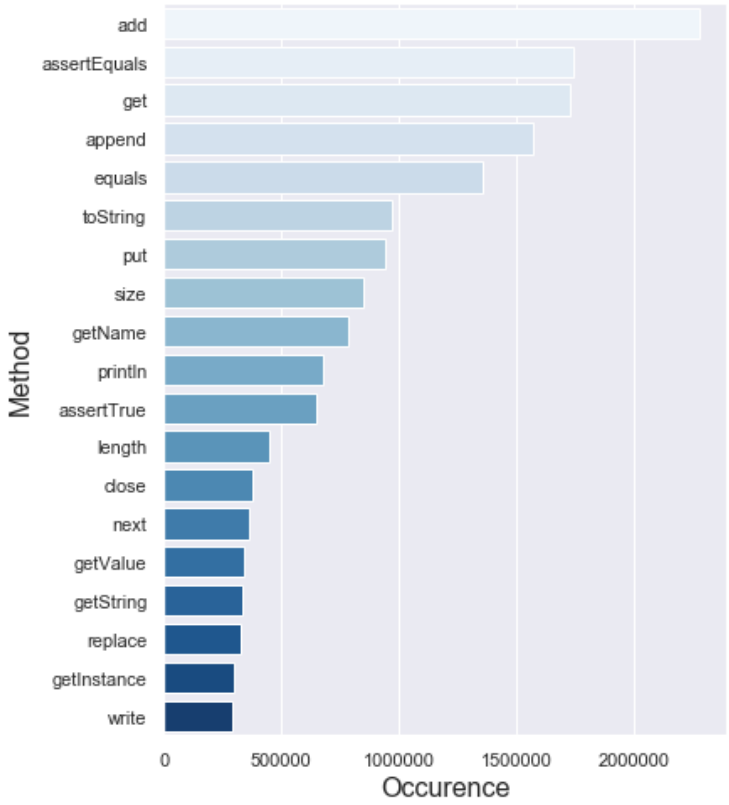



**Figure 4.3: Occurrences of the top-20 functions in the training corpus.**

In addition to this first training set, we also consider a variant of the data which consists of subtokens of the function names. We use this second training set in the Eclipse's suggestions reordering task and compare the results with the full function names alternative. Tokenizing the function names considerably reduces the size of the vocabulary. Furthermore, it is more likely that each subtoken of a function such as "*convertDateToString*" appears frequently in a corpus than the whole function name. Therefore, we expect that using subtokens can improve the completion results as compared to using the full names. Table **4.5** shows statistics of this second training set. We can observe that the number of types is significantly lower than in the first training set and that the minimum count parameter has almost no impact on the total number of tokens.

| | # Function sequences | Tokens | Types |
|---|---|---|---|
| *no min count* | 10.702.667 | 183.334.996 | 165.110 |
| *min count (5)* | 10.702.667 | 183.160.006 | 71.460 |

**Table 4.5: Training set 2 (*functions subtokens*).**

### 4.3.2 Evaluating the Paragraph Vector Model

As discussed in our approach, we consider two ways to evaluate the paragraph vector model on function completion tasks. The former is by building a function call completion system from scratch, and the latter is by reordering Eclipse's suggestions. Both strategies require different inputs that we describe in the next two subsections.

#### Function Completion from Scratch

For this experiment, we only evaluate the model with full functions sequences. For the evaluation, we follow Schnabel et al. [Schnabel et al., 2015] with an intrinsic evaluation that focuses on the relatedness aspect of the paragraph vectors (*RQ2*), and an extrinsic evaluation that measures the performance of the paragraph vector model on function completion tasks (*RQ3*).

In the intrinsic evaluation, we pick randomly several functions from the model vocabulary and compute their most similar functions. Then, we check whether close functions embeddings are consistent. We show relatedness results for common and specific functions. Additionally, we plot a t-SNE depicting clusters in the embeddings of the functions. For the extrinsic evaluation, for each method under test, we consider the sequence of calls preceded by the method name $(f_1, f_2, ..., f_c, f_{c+1}, ..., f_n)$. Then, we extract the context that will be used for the completion as defined in Section **4.2.2**, by cutting the sequence at a given call, say $f_c$. In that case, the context is of length $c$ and the call site to complete is $f_{c+1}$. We can sample many contexts by varying the size $c$, *i.e.,* the number calls before the call site to complete. For each test project, we iterate over the sequences of functions of its methods and extract contexts with size $c \in [1,8]$. Then, for each context we infer a paragraph vector and build a suggestion list to predict the $(c+1)$th function call. For the sake of evaluation, we fixed the maximum size of the suggestion lists to 10.

#### Reordering Eclipse's Suggestions

For this experiment, we first retrieve the suggestions of the Eclipse content assist plug-in for each call site of each test project. In Table **4.3**, *function calls* is the number of call sites per project, and then to the suggestion lists retrieved. To reorder these suggestion lists, we experiment with both full function names or subtokens strategies as explained in *Section **4.2.3***. For the evaluation with function name subtokens, instead of using the context as is, we tokenize it before inferring a vector representation. As for previous experiment, the maximum size of suggestion lists is 10 to allow comparison between all approaches.

### 4.3.3 Effectiveness Metrics

The evaluation aims to determine whether a paragraph vector model is able to efficiently provide good function call suggestion lists. To evaluate our systems, we consider that a set of suggestions is relevant if it reflects the user's need. That is, the suggestion list contains the correct function call that follows a given context.

To measure the relevance, we calculate two widely-used metrics, precision at $k$ (P@$k$) and the mean reciprocal rank (MRR). As there is a unique valid suggestion for each call site, P@$k$ for a test project is the number of times the expected function call appears in top-$k$ of suggestion lists divided by the number of tested call sites.

The second metric we report is MRR. The reciprocal rank is given by the inverse of the rank of the first relevant suggestion in the result of a test sample. Mean reciprocal rank for a test set $T$ is

$$MRR = \frac{1}{|T|}\sum_{i=1}^{|T|}\frac{1}{rank_i}$$

where $rank_i$ is the rank of the first relevant suggestion in the $i$-th test sample. For example, if on average, the relevant function call appears at rank 2, the MRR is 0.5.

## 4.4 Evaluation Results

In this section, we present the results of our experiments and answer the research questions. For the sake of clarity, we present the global results for questions $RQ3-4$ in Table **4.7** and illustrate them with 5 representative projects for each question.

### 4.4.1 Naturalness of Function Calls (RQ1)

Figure **4.4** shows the average cross-entropy on the 20 test projects including and excluding out-of-vocabulary (OOV) functions, *i.e.,* function names in the project that do not appear in the training set. The cross-entropy for the full functions is much higher than in Hindle et al.'s work. But it decreased by excluding OOV functions and it gets closer to the cross-entropy they reported on a Java corpus of ten projects. Furthermore, we observe that function names subtokens have a significantly lower cross-entropy and that excluding the OOV has no impact. The no decreasing of the cross-entropy when excluding OOV words means that almost all subtokens in the test projects appear in the training set. This means that sequences of functions subtokens are more predictable than sequences of full functions. We conclude that the naturalness hypothesis is more prevalent using subtokens of functions, but we may loose important information about the sharing of semantics across functions.

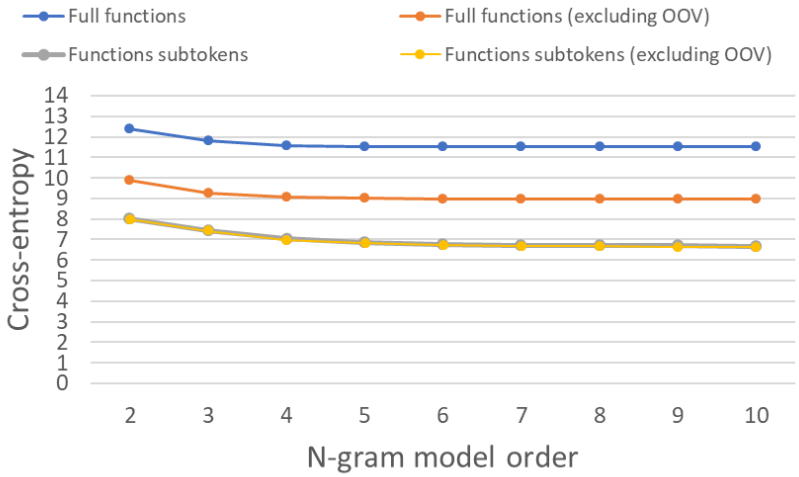



**Figure 4.4: Comparison of the cross-entropy on the 20 tests projects for full functions and subtokens functions with respect to the order of the $n$-gram model (RQ1).**

In their work, *Hindle et al.* estimated $n$-gram models on a Java corpus that includes all tokens present in the code. *Rahman et al.* addressed the same replication work and conclude that syntax tokens are much more present than identifiers in programming languages and that they make the code artificially predictable. The levels of cross-entropy that we report are closer than those reported in *Rahman et al.*'s work. That is, including only functions in the training set drastically decreases the predictability of the code.

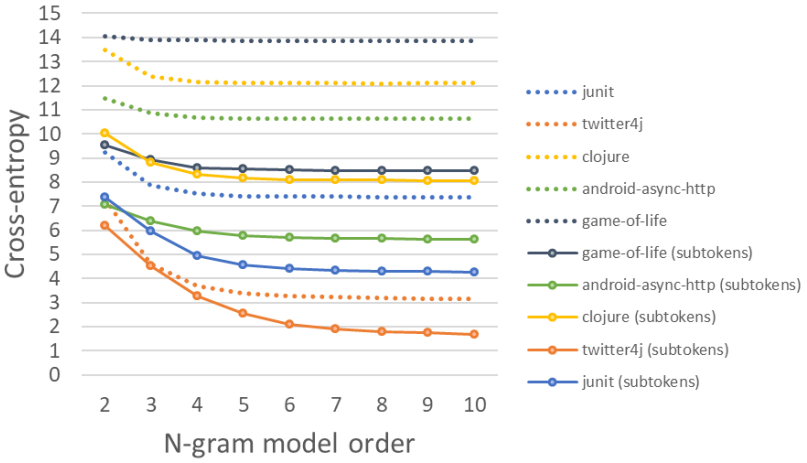



**Figure 4.5: Comparison of the cross-entropy for 5 tests projects with full functions and subtokens functions with respect to the order of the $n$-gram model (RQ1).**

In Figure **4.5** we report the cross-entropy on 5 test projects using full functions and subtokens functions. We can observe that some projects such as *twitter4j* and *junit* have very low cross-entropy, even when considering full functions. This can be explain by the

high vocabulary coverage of these two projects (*see Table **4.3***). In addition to that, we observe that the subtokens approach yields to a decreasing of the cross-entropy for all test projects.

RQ1 – Summary

We suspect that the paragraph vector model will perform well on projects that have a high vocabulary coverage and that the subtokens of functions approach could outperform the full functions approach, especially when the coverage of the test project is low.

### 4.4.2 Relatedness of Function Sequences (RQ2)

To address this research question, we first selected the 1,000 function names that the most frequently appear in the training dataset. Then we inspected a random sample of 100. For each, we looked at the top-3 most similar function names in the learned model to check if these are actually related to the considered function name. Table **4.6** above shows examples of the groups of names that we found. Our random inspection showed that the model captures semantic similarities between functions in a consistent way. That is, given a function, the most similar functions are semantically related, and it is reasonable to think that they appear in similar contexts in the code. This is particularly the case for domain-independent function names like those of the 5 first lines of the table. More interestingly, this observation also holds for domain-specific function names such as ones of the three last lines. For instance, for "*getKeyManagers*", the model is able to find similar functions related to the concept of encryption. In addition, we trained a t-SNE on a subpart of the first training set containing the 10.000 most frequent tokens. In Figure **4.6**, we plot 150 tokens and highlight clusters of functions related to the same high-level concept.

| Function | Top-3 Most Similar Functions |
|---|---|
| send | receive, getAddress, setReplyTo |
| exists | getAbsolutePath, delete, mkdir |
| getValue | getKey, entrySet, size |
| assertNotNull | assertTrue, assertNull, assertEquals |
| readUnsignedShort | readUTF8, readClass, readUnsignedByte |
| getImage | setImage, createImage, getColumnImage |
| deleteNode | copyNode, createNewNode, getNodeById |
| getKeyManagers | getTrustManagers, getDefaultAlgorithm, createSSLContext |

**Table 4.6: Example of function relatedness captured by the embedding model (RQ2).**

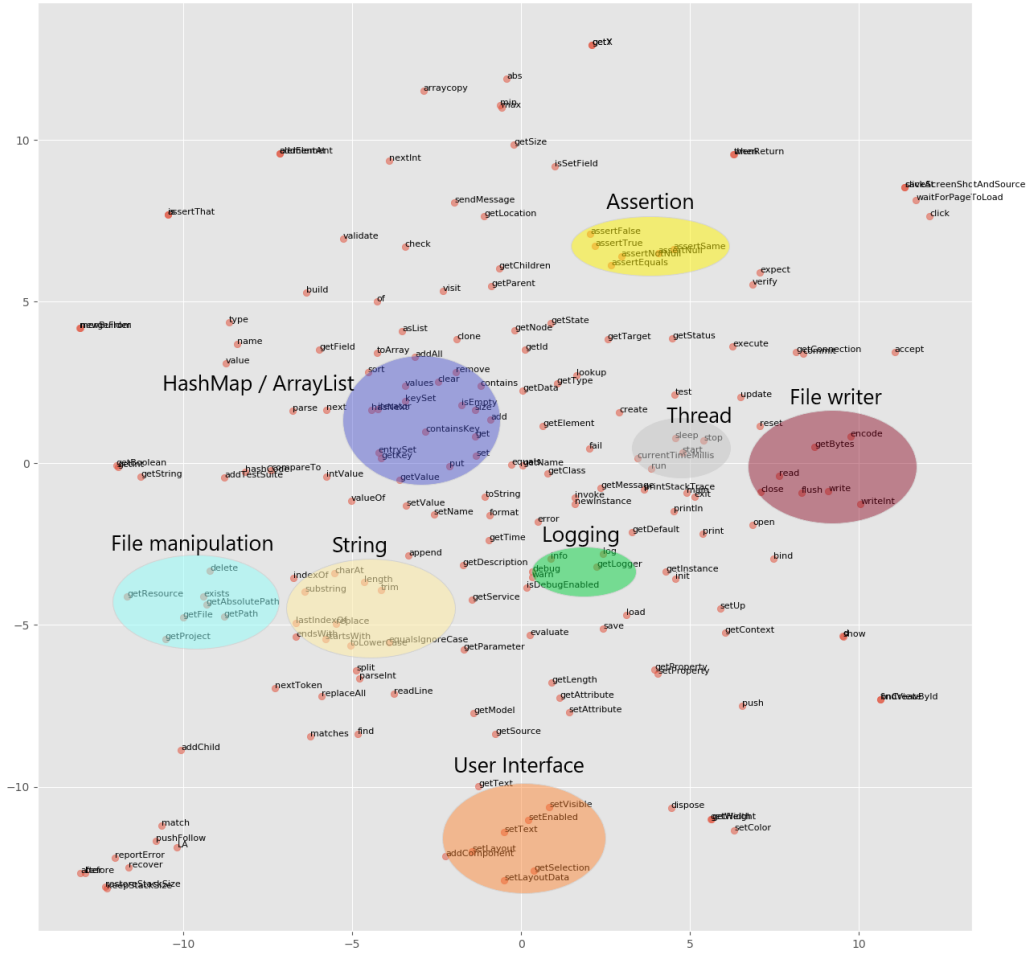



**Figure 4.6: t-SNE of the full methods paragraph vector model showing clusters of related function calls (RQ2).**

RQ2 – Summary

Considering the large size of the training set, we can conclude that there exist regularities in the concepts of the corpus and that our model is capable of capturing them. Thus, we do believe that paragraph vector embedding models can capture high-level concepts from the code.

### 4.4.3 Function Completion using Paragraph Vector Model (RQ3)

From the global results in Table **4.7**, we explain in more details the results for Eclipse and Doc2vec. As we can see, both columns are in general close, but Doc2vec completions are less good, as compared to those of Eclipse completion. Except for *twitter4j* and, to a lesser extent, *android-async-http* and *junit*, the Precision@10 (*P@10*) is lower. This is understandable as this completion strategy does not take into account the list of functions that can be called in the tested project.

These results are still interesting, especially when we consider the size of the context used for the completion. Indeed, in Figure **4.7**, we present the evolution of the Precision@10 according to the size of the contexts sampled for 5 tests projects. The curves show that greater contexts increase drastically the precision for all projects. This is explainable because when we provide a large context, the model is able to find more precise sequences of functions in terms of similarity. Fluctuations for *game-of-life* can be explained by the small size of the project and the lowest percentage of vocabulary coverage (*64%*). Furthermore, as we suspected in Section **4.4.1**, projects with the highest vocabulary coverage have the highest P@10. This is especially true for *twitter4j* with P@10 reaching 0.8 for size of contexts of 6.

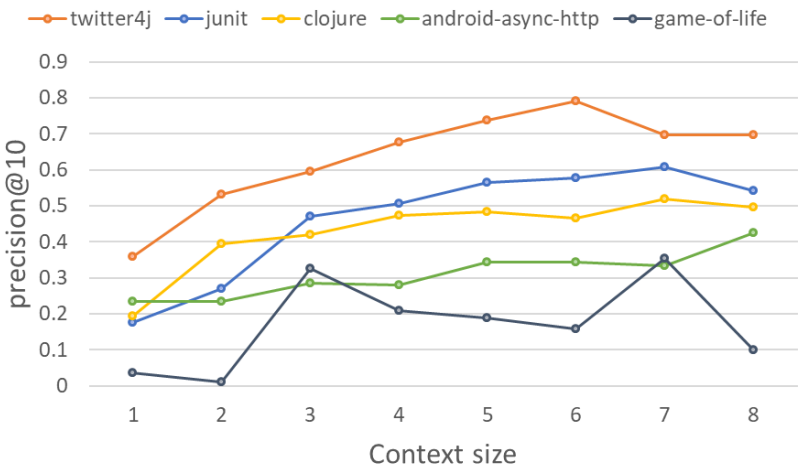



**Figure 4.7: Evolution of Precision@10 for 5 tests projects with respect to the size of the sampled contexts for function completion (RQ3).**

Figure **4.8** shows the evolution of P@k for this particular test project *w.r.t.* to the size of the context. As we can observe, the precision starts to saturate at P@10 which also shows that the choice of P@10 for evaluating our systems is reasonable.

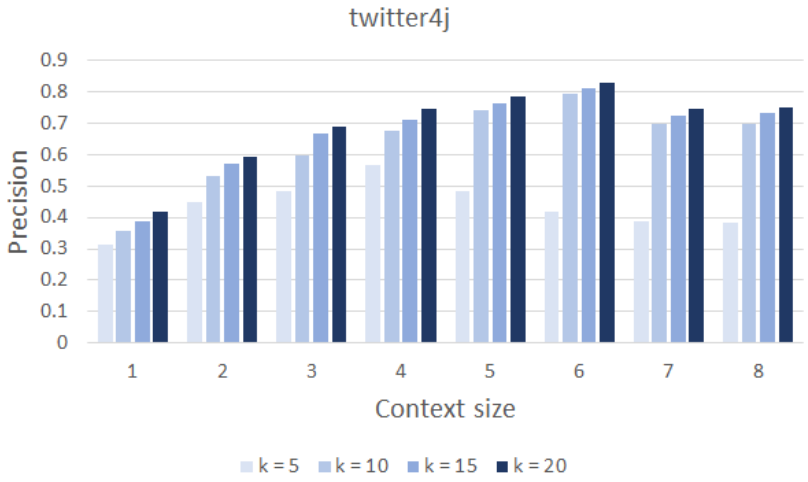



**Figure 4.8: Comparison of the Precision@k for *twitter4j* with the increasing of the size of the context (RQ3).**

Figure **4.9** summarizes the results and includes the mean reciprocal rank (MRR) for the 5 projects. We observe that for most projects the MRR is between 0.1 and 0.22, which means that the relevant suggestion appears, on average, between the fourth and the tenth position in the list.

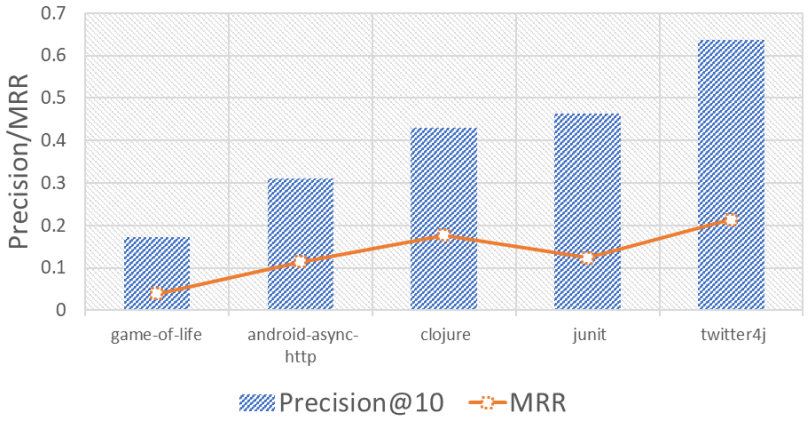



**Figure 4.9: Average Precision@10 and MRR on 5 test projects for function completion task (RQ3).**

RQ3 – Summary

In conclusion, although the performance of this completion strategy is close to one of Eclipse, and that the precision increases with the size of the context, we cannot state that using a paragraph-vector model for call completion is sufficient without considering the specific situation of the project under development.

### 4.4.4 Extending Eclipse's Content Assist (RQ4)

We compare the reordering performance of both paragraph vector models (*i.e., with full function names and subtokens*) with Eclipse content assist plug-in. As we can see in Table **4.7**, the strategy with full names outperforms clearly the Eclipse baseline both for the Precision@10 and the MRR. The only exception is *game-of-life* for which the model with subtokens improved slightly the precision of Eclipse. This means that the subtokens model can be a substitution solution when the coverage of function names by the model is low.

Figure **4.10** summarizes the scores of Precision@10 and MRR for the three completion strategies on the same 5 test projects. An interesting observation is the big improvement in Precision@k and MRR brought by the full-names model for *twitter4j* and, in MRR for *clojure*. This can be explained by the high vocabulary coverage in these projects (*respectively 99% and 94%*). However, in Figure **4.5**, *clojure* has also a very high cross-entropy meaning that the sequences of functions in the project are difficult to predict. Despite this, our model is able to find useful similar function sequences to perform accurate completions. The MRR above 0.5 for *twitter4j* and *clojure* indicates that the correct function suggestion appears between the first and second rank in the list, on average.

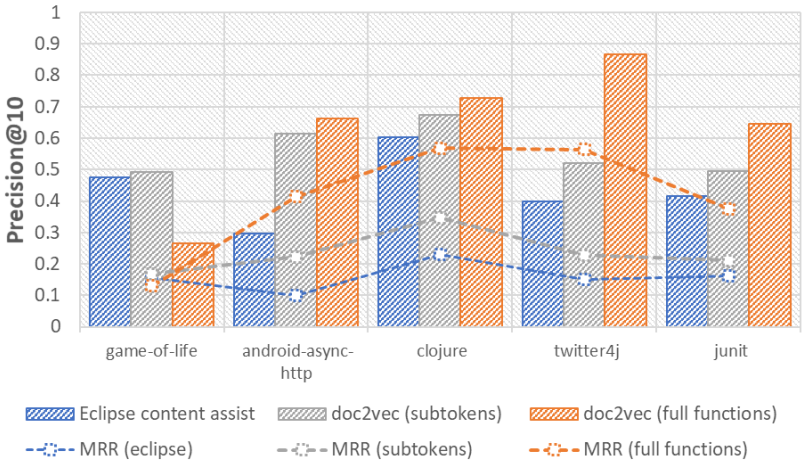



**Figure 4.10: Comparison of Precision@10 and MRR for the three systems on 5 test projects for Eclipse's suggestions reordering task (RQ4).**

Finally, another aspect that we evaluated is the time to produce completion suggestions for a call site. This time is on average between 700 ms and 800 ms, which makes our approach usable in a real programming setting.

RQ4 – Summary

In conclusion, based of the large number of tested call sites, we can state that using the paragraph vector model with full-function names to reorder Eclipse suggestions improved dramatically the correctness of Eclipse's content assist plug-in without a negative impact on the response time.

### Global Results (RQ3 and RQ4)

| | | Eclipse | | Doc2vec | | Eclipse + Doc2vec | | Eclipse + Doc2vec (subtokens) | |
|---|---|---|---|---|---|---|---|---|---|
| **Project** | **Size** | P@10 | MRR | P@10 | MRR | P@10 | MRR | P@10 | MRR |
| game-of-life | 128 | 0.4766 | **0.1786** | 0.1721 | 0.0388 | 0.2659 | 0.1298 | **0.4922** | 0.168 |
| android-async-http | 675 | 0.2966 | 0.1134 | 0.3097 | 0.1144 | **0.6637** | **0.4132** | 0.6133 | 0.2218 |
| clojure | 13020 | 0.6028 | 0.2391 | 0.4306 | 0.1775 | **0.729** | **0.5687** | 0.6752 | 0.3478 |
| twitter4j | 13365 | 0.3996 | 0.1659 | 0.636 | 0.2138 | **0.8665** | **0.5642** | 0.5209 | 0.2277 |
| facebook-android-sdk | 5689 | 0.4379 | 0.1688 | 0.2198 | 0.0921 | 0.504 | **0.2906** | **0.5168** | 0.2174 |
| hystrix | 5790 | 0.2038 | 0.0831 | 0.1846 | 0.079 | **0.4789** | **0.3183** | 0.4472 | 0.1817 |
| junit | 8144 | 0.4169 | 0.1819 | 0.464 | 0.1237 | **0.6444** | **0.3752** | 0.4946 | 0.2115 |
| antlr | 11053 | 0.3561 | 0.1411 | 0.2889 | 0.112 | **0.5916** | **0.3358** | 0.4989 | 0.1817 |
| mongo-java-driver | 35836 | 0.3734 | 0.1519 | 0.2387 | 0.1235 | **0.4937** | **0.2955** | 0.4268 | 0.1713 |
| netty | 72754 | 0.3258 | 0.1326 | 0.2303 | 0.1041 | **0.4782** | **0.2649** | 0.3914 | 0.1546 |

**Table 4.7: Global results of the experiments (RQ3 and RQ4).**

### 4.4.5 Threats to Validity

We identified some threats to the validity and attempted to address them during the design of our evaluation. The first threat relates to the mono-operation bias as we experimented only with Java projects. We conjecture that our approach can be used for call completion in other languages as we do not rely on Java language constructs, but on identifiers. To prevent the mono-method bias, we evaluated our approach with two metrics commonly used to measure the effectiveness of ranking systems. Another threat concerns the interaction of setting and treatment. Indeed, we reused and compared our results with the completion in Eclipse. It has been shown that Eclipse content assist plug-in is commonly used by Java developers [Murphy et al., 2006], and we do believe that it is representative enough. Another important aspect that we considered is the representativeness of the dataset. We made sure to train our models on a large dataset of open-source projects from different domains of application and of variable sizes. For the evaluation, we choose a variety of test projects as well. Finally, an important threat to the validity of our results

arises from the choice of hyper-parameters of the paragraph vector models. To address the issue, we followed guidelines from the literature. We tuned the hyper-parameters that influence the most the quality of embeddings[2] and chose commonly used values for the other hyper-parameters, following Lau and Baldwin's recommendations [Lau and Baldwin, 2016]. The model used in the experiments is a PV-DBOW with dimension of embeddings of 300, a window size of 15, a threshold of 20 for minimum word counts and hierarchical softmax as training algorithm.

## 4.5 Conclusion

**Conclusion**

Regarding the RQs, we summarize the results as follows :

- **RQ1**: We were able to replicate *Hindle et al.* and *Rahman et al.* works by showing that functions are highly unpredictable and responsible for the high-level of cross-entropy in the code. Moreover, we reported this level of cross-entropy for 5 test projects depicting the relationship between the projects' vocabulary coverage and their level of cross-entropy.
- **RQ2**: By performing an intrinsic evaluation of our paragraph vector model, we showed that there exist regularities across the projects of the training corpus and conclude that our model is able to capture high-level concepts from the code. Thus, we answer to this RQ in the affirmative.
- **RQ3**: Even though our model is able to suggest relevant function calls, the results are generally less good than Eclipse content assist, except for a few test projects. We conclude that our paragraph vector model is not sufficient by itself without considering the possible calls given the project under development. Therefore, we answer negatively to this RQ.
- **RQ4**: Combining our paragraph vector model with Eclipse content assist plug-in improved the correctness of the recommendations for 9 out of the 10 test projects, by up to 135% reaching 85% of Precision@10. We also highlighted the fact that projects with high vocabulary coverage have the best results in term of P@10 and MRR. Thus, we answer positively to this RQ.

[2] https://code.google.com/archive/p/word2vec

# CHAPTER 5

# FUTURE WORK

Our approach is general enough to be used for other related tasks with small adaptations. In Section **5.1**, we start this chapter by identifying a few ways to improve our approach for function-call completion. In Section **5.2**, we introduce opportunities in code search, an important feature for developers, which is very related to code completion. Finally, we present how embedding-based models could help a modeler to design diagrams.

## 5.1 Extension of Our Approach

The fact that our approach is not based on the structure of the language make it usable for any programming language. Indeed, identifiers are generally common to the programming languages and thus we conjecture that our model can be used for function-call completion in many of these. We identified three ways to evaluate our approach for several programming languages :

1. **Transfer learning**. We evaluate the model that we presented in this work directly on several other programming languages. The full function model might not be testable for some languages due to the specific structure of the identifiers in Java (*i.e., camel case*). However, the subtoken model is more general and can be tested on any language.
2. **Cross-language model**. We train a paragraph vector model based on a dataset made of tokenized identifiers from several programming languages.
3. **Independent models**. We train as many models as the programming languages we have in our data. Each model is then evaluated on its corresponding programming

language. Such evaluation would allow us to determine whether identifiers in some languages are easier to predict or not.

The first two methods are more desirable since they require only one model to be trained and thus less storage on a physical disk. On the other hand, the transfer learning method does not require the model to be re-trained, but since the training is performed offline, the cost of training a cross-language or independent models is only about gathering more data and training time.

The second opportunity that we identified is to evaluate our approach with deep learning models and state-of-the-art models in NLP. In this work, we have considered code completion as a similarity task where the call site context is matched with contexts of thousands of projects. As discussed in Chapter **3**, most of the related work in code completion use generative models such as language models to perform the completion. We intend to explore these kind of models and evaluate them for function-call completion.
Besides, we also intend to evaluate our approach using wider scopes than method declarations. Models such as LSTM and attention-based neural networks would allow us to capture long-range dependencies between identifiers in the code (*i.e., within a class or a module*) and improve our system.

## 5.2 Code Search

As well as code completion, code search is crucial for developers. Such feature is used by developers in many ways to better understand snippets of code or to find out how to implement something [Sadowski et al., 2015].
Code search is an information retrieval (**IR**) research field that have adapted techniques traditionally used for search engines. It requires a set of document and a set of queries at one's disposal. The documents are usually snippets of code (*e.g., a method declaration and its body*) and the query are designed to represent a user need. We distinguish two main approaches to perform the code search :

1. **Traditional approach.** Each document is labelled with a relevance judgement that denotes its relevance *w.r.t* each query. The code search system retrieves documents for each query and is evaluated using typical IR metrics.
2. **Full relevance feedback.** Given a query, the code search system retrieves code snippets. The user of the system identify the relevant code snippets and inform the system which updates itself in order to produce better rankings.

The traditional approach have been explored in the literature using traditional IR models, learning-to-rank or colloborative-filtering techniques [Holmes et al., 2005; Nguyen et al.,

2019; Niu et al., 2016]. The most recent approaches have focused on the utilization of deep learning models and are very related to ours for code completion [Cambronero et al., 2019; Gu et al., 2018; Husain et al., 2019].
In fact, unsupervised machine learning using document embeddings models such as *Doc2vec* can be used for code search. The learning phase would be very similar to ours and consists of learning embeddings of code snippets. Then, we embed the queries in the embedding space and retrieve the most similar code snippets. Thus, with a dataset of code snippets and queries our approach could be used without requiring any adaptation.

On the other hand, techniques based on relevance feedback for code search require more in-depth explorations [Wang et al., 2014]. We intend to work on the code search problematic by mining Stack Overflow posts and compare traditional approaches with full relevance feedback techniques.

## 5.3 Diagram Completion

The last main opportunity that we identified for future work is about diagram completion. To our knowledge, the development of such an approach has not yet been explored in the literature and we conjecture that a diagram completion tool could become an important feature in diagram design environments. In fact, such a tool would help the modeler to design diagram by providing naming suggestions or subparts of the diagram.
Similarly to the code, a diagram can be represented as a sequence of tokens. But, the main issue of such representation is that a diagram is not sequential. Moreover, the syntax and semantics of UML diagrams is quiet rich and complex. Therefore, this is an exiting and open challenge that requires a critical diagram feature engineering step in order to support the learning process and provide meaningful recommendations to the modeler.

# CHAPTER 6

# CONCLUSION

In this thesis, we presented an approach for function-call completion that can be used alone or integrated with a code completion tool based on a language typing system. Our approach starts from the assumption that it is possible to abstract application-independent high-level concepts in the form patterns of call sequences contained in code repositories. To this end, we built on document-embedding algorithms to train models that can be exploited for function-call completion. Our experiments highlight promising results for most of the tested projects and indicate that our trained model captures useful high-level concepts that can be used for completion. This shows that our approach can be useful for helping developers writing their software even for new projects and with limited knowledge about the used APIs.

Although the obtained results are satisfactory, there is room for improvement. One of the limitations of our approach is that it has a limited efficiency with projects having very specific function names, not frequent in existing code repositories. We plan to improve the natural-language processing pipeline to cope with this situation. We also plan to explore other embedding-based language models to improve the completion. Finally, instead of capturing high-level concepts inside a method scope, we plan to learn similar concepts in wider scopes and thus learning recurring long-range dependencies that could be useful for program summarizing.

From another perspective, the fact that our approach does not rely on language constructs, but rather on sequences of identifiers used in method names opens the door for many other possibilities to explore. Indeed, we conjecture that the learned models can be reused cross-programming languages. They can also be used, with some adaptation, to assist developers for other tasks such as program documentation by providing summaries,

construct naming for automated generation, clone detection, and code search. Finally, an approach similar to ours can be employed to assist in building design diagrams such those of UML.


# APPENDIX A

## A.1 Approach

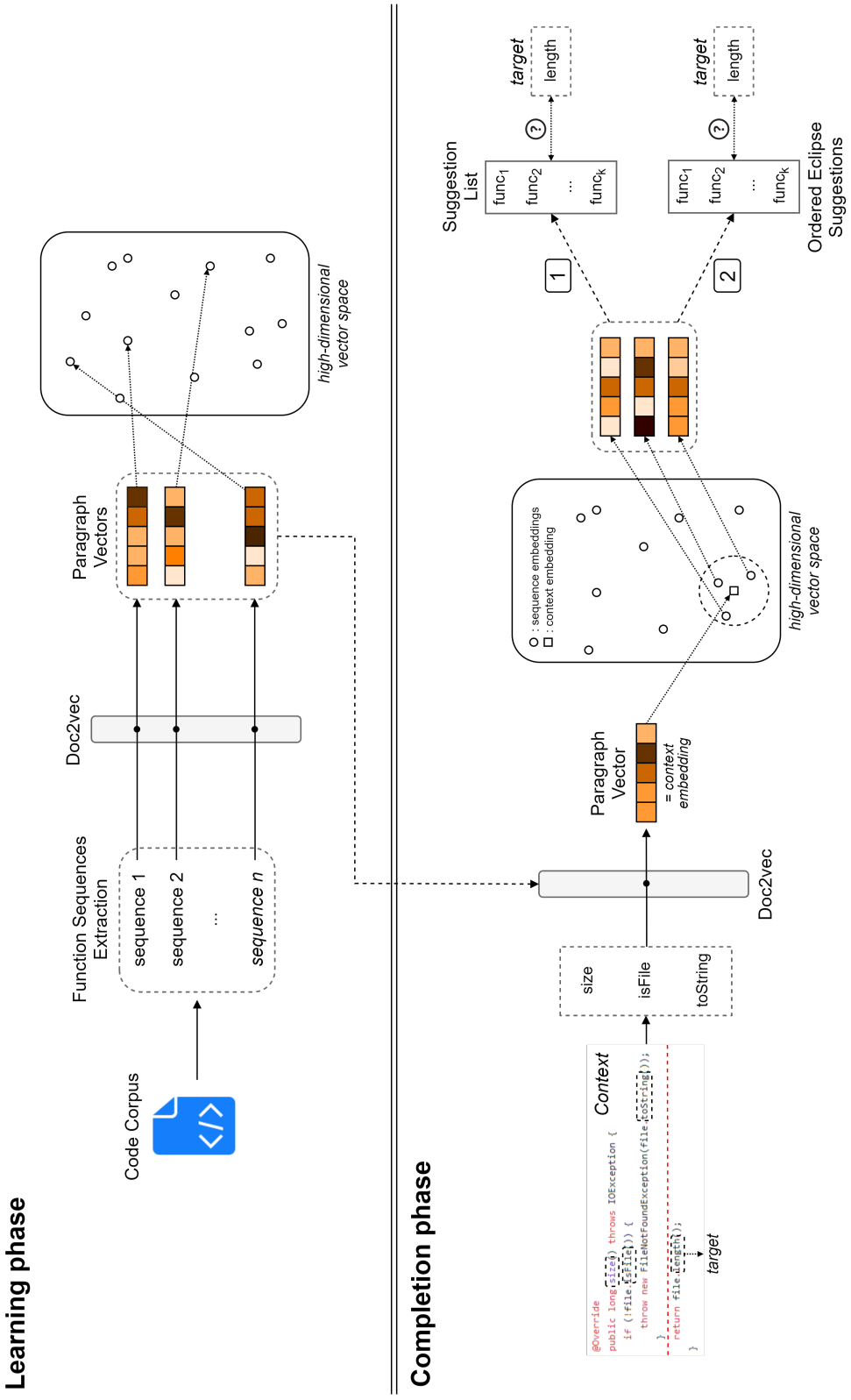



**Figure A.1: Approach**

# BIBLIOGRAPHY


Allamanis, M., Barr, E. T., Bird, C., and Sutton, C. (2015).
Suggesting accurate method and class names.
In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 38–49, New York, NY, USA. Association for Computing Machinery.

Allamanis, M., Barr, E. T., Devanbu, P. T., and Sutton, C. A. (2017).
A survey of machine learning for big code and naturalness.
*CoRR*, abs/1709.06182.

Allamanis, M., Peng, H., and Sutton, C. A. (2016).
A convolutional attention network for extreme summarization of source code.
*CoRR*, abs/1602.03001.

Allamanis, M. and Sutton, C. (2013a).
Mining source code repositories at massive scale using language modeling.
In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 207–216.

Allamanis, M. and Sutton, C. (2013b).
Mining source code repositories at massive scale using language modeling.
In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 207–216.

Alon, U., Sadaka, R., Levy, O., and Yahav, E. (2019).
Structural language models of code.

Bahdanau, D., Cho, K., and Bengio, Y. (2014).
Neural machine translation by jointly learning to align and translate.

Bengio, Y., Ducharme, R., Vincent, P., and Janvin, C. (2003).
A neural probabilistic language model.
*J. Mach. Learn. Res.*, 3(null):1137–1155.

Bhoopchand, A., Rocktäschel, T., Barr, E. T., and Riedel, S. (2016).
Learning python code suggestion with a sparse pointer network.
*CoRR*, abs/1611.08307.

Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. (2016).
Enriching word vectors with subword information.
*CoRR*, abs/1607.04606.

Bruch, M., Monperrus, M., and Mezini, M. (2009).
Learning from examples to improve code completion systems.
In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, page 213–222, New York, NY, USA. Association for Computing Machinery.

Cambronero, J., Li, H., Kim, S., Sen, K., and Chandra, S. (2019).
When deep learning met code search.
*CoRR*, abs/1905.03813.

Chen, S. F. and Goodman, J. (1996).
An empirical study of smoothing techniques for language modeling.
In *34th Annual Meeting of the Association for Computational Linguistics*, pages 310–318, Santa Cruz, California, USA. Association for Computational Linguistics.

Chen, Z. and Monperrus, M. (2019).
A literature study of embeddings on source code.
*CoRR*, abs/1904.03061.

Dai, A. M., Olah, C., and Le, Q. V. (2015).
Document embedding with paragraph vectors.
*CoRR*, abs/1507.07998.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. (2018).
BERT: pre-training of deep bidirectional transformers for language understanding.
*CoRR*, abs/1810.04805.

Goodman, J. (2001).
A bit of progress in language modeling.
*CoRR*, cs.CL/0108005.

Gu, X., Zhang, H., and Kim, S. (2018).
Deep code search.
In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944.

Hashimoto, K., Kontonatsios, G., Miwa, M., and Ananiadou, S. (2016).
Topic detection using paragraph vectors to support active learning in systematic reviews.

*J. of Biomedical Informatics*, 62(C):59–65.

Heafield, K. (2011).
Kenlm: Faster and smaller language model queries.

Hellendoorn, V. J. and Devanbu, P. (2017).
Are deep neural networks the best choice for modeling source code?
In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 763–773, New York, NY, USA. Association for Computing Machinery.

Hindle, A., Barr, E. T., Su, Z., Gabel, M., and Devanbu, P. (2012).
On the naturalness of software.
In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 837–847. IEEE Press.

Hochreiter, S. and Schmidhuber, J. (1997).
Long short-term memory.
*Neural Comput.*, 9(8):1735–1780.

Holmes, R., Walker, R. J., and Murphy, G. C. (2005).
Strathcona example recommendation tool.
*SIGSOFT Softw. Eng. Notes*, 30(5):237–240.

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. (2019).
Codesearchnet challenge: Evaluating the state of semantic code search.

Ismailov, A., Jalil, M. A., Abdullah, Z., and Rahim, N. A. (2016).
A comparative study of stemming algorithms for use with the uzbek language.
In *2016 3rd International conference on computer and information sciences (ICCOINS)*, pages 7–12. IEEE.

Karampatsis, R.-M., Babii, H., Robbes, R., Sutton, C., and Janes, A. (2020).
Big code != big vocabulary: Open-vocabulary models for source code.

Kim, S., Zhao, J., Tian, Y., and Chandra, S. (2020).
Code prediction by feeding trees to transformers.

Kneser, R. and Ney, H. (1995).
Improved backing-off for m-gram language modeling.
In *1995 International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages 181–184 vol.1.

Lau, J. H. and Baldwin, T. (2016).
An empirical evaluation of doc2vec with practical insights into document embedding generation.
*CoRR*, abs/1607.05368.

Le, Q. V. and Mikolov, T. (2014).
Distributed representations of sentences and documents.
*CoRR*, abs/1405.4053.

Li, J., Wang, Y., King, I., and Lyu, M. R. (2017).
Code completion with neural attention and pointer networks.
*CoRR*, abs/1711.09573.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a).
Efficient estimation of word representations in vector space.
*CoRR*, abs/1301.3781.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. (2013b).
Distributed representations of words and phrases and their compositionality.
*CoRR*, abs/1310.4546.

Mikolov, T., Yih, W.-t., and Zweig, G. (2013c).
Linguistic regularities in continuous space word representations.
In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 746–751, Atlanta, Georgia. Association for Computational Linguistics.

Murphy, G. (2019).
Beyond integrated development environments: Adding context to software development.
pages 73–76.

Murphy, G., Kersten, M., and Findlater, L. (2006).
How are java software developers using the eclipse ide?
*IEEE Software*, 23:76–83.

Nguyen, A. T. and Nguyen, T. N. (2015).
Graph-based statistical language model for code.
In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, page 858–868. IEEE Press.

Nguyen, P. T., Di Rocco, J., Di Ruscio, D., Ochoa, L., Degueule, T., and Di Penta, M. (2019).
Focus: A recommender system for mining api function calls and usage patterns.

In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1050–1060.

Nguyen, T. T., Nguyen, A. T., Nguyen, H. A., and Nguyen, T. N. (2013).
A statistical semantic language model for source code.
In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 532–542, New York, NY, USA. Association for Computing Machinery.

Niu, H., Keivanloo, I., and Zou, Y. (2016).
Learning to rank code examples for code search engines.
*Empirical Software Engineering*, 22.

Pennington, J., Socher, R., and Manning, C. (2014).
GloVe: Global vectors for word representation.
In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar. Association for Computational Linguistics.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018).
Deep contextualized word representations.
*CoRR*, abs/1802.05365.

Porter, M. F. (1980).
An algorithm for suffix stripping.
*Program*, 40:211–218.

Proksch, S., Lerch, J., and Mezini, M. (2015).
Intelligent code completion with bayesian networks.
*ACM Trans. Softw. Eng. Methodol.*, 25(1).

Rahman, M., Palani, D., and Rigby, P. (2019).
Natural software revisited.

Raychev, V., Vechev, M., and Yahav, E. (2014).
Code completion with statistical language models.
In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 419–428, New York, NY, USA. Association for Computing Machinery.

Rong, X. (2014).
word2vec parameter learning explained.
*CoRR*, abs/1411.2738.

Sadowski, C., Stolee, K. T., and Elbaum, S. (2015).
How developers search for code: A case study.
In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 191–201, New York, NY, USA. Association for Computing Machinery.

Schnabel, T., Labutov, I., Mimno, D., and Joachims, T. (2015).
Evaluation methods for unsupervised word embeddings.
In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 298–307, Lisbon, Portugal. Association for Computational Linguistics.

Svyatkovskiy, A., Lee, S., Hadjitofi, A., Riechert, M., Franco, J., and Allamanis, M. (2020).
Fast and memory-efficient neural code completion.

Svyatkovskiy, A., Zhao, Y., Fu, S., and Sundaresan, N. (2019).
Pythia: Ai-assisted code completion system.
*Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*.

Tu, Z., Su, Z., and Devanbu, P. (2014).
On the localness of software.
In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 269–280, New York, NY, USA. Association for Computing Machinery.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017).
Attention is all you need.
*CoRR*, abs/1706.03762.

Wang, S., Lo, D., and Jiang, L. (2014).
Active code search: Incorporating user feedback to improve code search relevance.
In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 677–682, New York, NY, USA. Association for Computing Machinery.